ALGORITHMIC TECHNIQUES FOR NANOMETER VLSI DESIGN AND MANUFACTURING CLOSURE

A Dissertation

by

SHIYAN HU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2008

Major Subject: Computer Engineering

ALGORITHMIC TECHNIQUES FOR NANOMETER VLSI DESIGN AND MANUFACTURING CLOSURE

A Dissertation

by

SHIYAN HU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Jiang Hu
Committee Members,	Charles J. Alpert
	Mosong Cheng
	Donald K. Friesen
	Weiping Shi
Head of Department,	Costas N.Georghiades

May 2008

Major Subject: Computer Engineering

ABSTRACT

Algorithmic Techniques for Nanometer VLSI Design and Manufacturing Closure.

(May 2008)

Shiyan Hu, B.S., Beijing University of Aeronautics and Astronautics;

M.S., Polytechnic University, Brooklyn, NY

Chair of Advisory Committee: Dr. Jiang Hu

As Very Large Scale Integration (VLSI) technology moves to the nanoscale regime, design and manufacturing closure becomes very difficult to achieve due to increasing chip and power density. Imperfections due to process, voltage and temperature variations aggravate the problem. Uncertainty in electrical characteristic of individual device and wire may cause significant performance deviations or even functional failures. These impose tremendous challenges to the continuation of Moore's law as well as the growth of semiconductor industry.

Efforts are needed in both deterministic design stage and variation-aware design stage. This research proposes various innovative algorithms to address both stages for obtaining a design with high frequency, low power and high robustness. For deterministic optimizations, new buffer insertion and gate sizing techniques are proposed. For variation-aware optimizations, new lithography-driven and post-silicon tuning-driven design techniques are proposed.

For buffer insertion, a new slew buffering formulation is presented and is proved to be NP-hard. Despite this, a highly efficient algorithm which runs $> 90\times$ faster than the best alternatives is proposed. The algorithm is also extended to handle continuous buffer locations and blockages.

For gate sizing, a new algorithm is proposed to handle discrete gate library in contrast to unrealistic continuous gate library assumed by most existing algorithms.

Our approach is a continuous solution guided dynamic programming approach, which integrates the high solution quality of dynamic programming with the short runtime of rounding continuous solution.

For lithography-driven optimization, the problem of cell placement considering manufacturability is studied. Three algorithms are proposed to handle cell flipping and relocation. They are based on dynamic programming and graph theoretic approaches, and can provide different tradeoff between variation reduction and wirelength increase.

For post-silicon tuning-driven optimization, the problem of unified adaptivity optimization on logical and clock signal tuning is studied, which enables us to significantly save resources. The new algorithm is based on a novel linear programming formulation which is solved by an advanced robust linear programming technique. The continuous solution is then discretized using binary search accelerated dynamic programming, batch based optimization, and Latin Hypercube sampling based fast simulation.

To my parents Changxin Hu and Xiaoyu Hu.

ACKNOWLEDGMENTS

I would like to express my great thanks to my advisor Dr. Jiang Hu for his kind guidance for my Ph.D. study. Dr. Jiang Hu introduced me the field of VLSI Computer-Aided Design. He shared his deep knowledge and research experience with me and constantly provided invaluable advise to me. I truly appreciate all of his academic, moral and financial support to me.

Many thanks to my Ph.D. dissertation committee members, Dr. Charles Alpert, Dr. Mosong Cheng, Dr. Donald Friesen, Dr. Jiang Hu and Dr. Weiping Shi. I really appreciate their invaluable assistance to my dissertation.

In addition, I would like to thank Dr. Weiping Shi for instructing great courses on physical design where I learned a lot. He also spent much time in discussing various CAD problems with me. I really appreciate Dr. Charles Alpert in IBM Austin Research Lab for being my mentor and manager when I was an intern there. He shared his great academic and industrial experience with me. Special thanks to Dr. Mosong Cheng and Dr. Donald Friesen for giving many highly valuable comments on my preliminary examination, proposal and dissertation. I would also like to thank the graduate students Ganesh Venkataraman, Zhuo Feng, Pratik Shah, Zhanyuan Jiang and Nikhil Jayakumar in Computer Engineering group at Texas A&M University for their helps on my research.

Last, but not least, I would like to express my greatest gratefulness to my family for their long-lasting encouragement and support.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

TABLE Page

TABLE Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

A. Preliminaries and Motivation

As Very Large Scale Integration (VLSI) technology moves to the nanoscale regime, feature size keeps shrinking which results in large chip density (refer to Table I). Consequently, a chip often consists of millions of gates and circuit design becomes increasingly complex. Design automation techniques are essential to meet the challenge of tightening time-to-market pressure and shortening semiconductor product cycles.

Table I. Technology trend for VLSI chips [1].

Technology node	$130nm$	$90nm$	$65nm$	$45nm$
Chip density ($\frac{Million\ transistors}{cm^2}$)	25	77	154	309
Gate length (nm)	90	53	25	18
Tolerable variation (nm)	5.3	3.75	2.5	2
Wavelength (nm)	248	193	193	193

In spite of the advances in design automation techniques, design closure is increasingly difficult to achieve. A critical issue is to close the timing with low power consumption. Prevailing Integrated Circuit (IC) designs often have higher power dissipation than before due to greater device density, more metal layers and faster clock frequency, etc. This trend can be seen from Figure 1 which shows the power consumption of a class of system-on-ship designs. High power consumption has been a crucial concern for both portable electronics and non-mobile systems. For portable

---

The journal model is *IEEE Transactions on Automatic Control.*

electronics, low power dissipation may elongate battery lifetime. For non-mobile systems, large amount of heat due to high power dissipation imposes great difficulties to the packaging and cooling systems. Clearly, both circuit performance and reliability substantially depend on the solution to the power-heat problem.

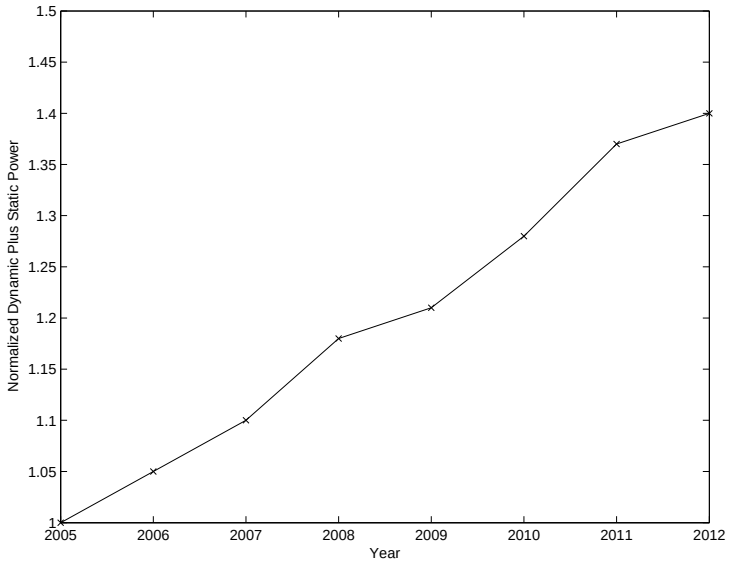


Fig. 1. The power consumption for a system-on-chip design [1].

Power issues must be addressed in circuit design. For example, when performing buffer insertion to the circuit for timing and slew optimization, a significant part of the performance of the design depends on using as little buffering resources as possible since buffers themselves are a drain on power and can cause other gates to be sized to higher power levels. As power can be modeled by a generic cost function, cost minimization needs to be considered during circuit optimization. This dissertation explores two widely used optimization techniques, namely buffer insertion and gate sizing, and proposes several innovative algorithmic techniques for performance optimization with cost minimization.

On the other hand, with fast technology scaling, imperfections due to process,

voltage and temperature variations impose a tremendous challenge to the continuation of Moore's law as well as the growth of semiconductor industry. With increasingly shrinking features on the die, uncertainty in electrical characteristic of each individual device and wire significantly increases. Consequently, circuit performance is no longer determined solely by deterministic values and many previously negligible variations start to manifest, which may cause performance deviations or even functional failures. Even for $180nm$ technology, variations up to $20\times$ in leakage and 30% in frequency have been reported in industrial chip designs [2].

Since variations may significantly divert circuit performance from the desired values, optimizations need to be performed to mitigate variation effects to achieve design and manufacturing closure. One way is to perturb the design to mitigate the variations it may receive. For example, an existing layout can be modified to be lithography friendly and then variations can be significantly reduced in fabrication process. Since there are always variations even after strongest variation mitigation techniques, it is also necessary to enable a circuit to be tunable after fabrication to further compensate for variations. These two classes of techniques are complementary to each other and this dissertation explores both of them to address the variation issues.

Clearly, to achieve design and manufacturing closure, efforts are needed in both deterministic design stage and variation-aware design stage. Deterministic design stage refers to the traditional circuit design without handling variations. With increasing design complexity, innovative effective and efficient algorithmic techniques are needed to compute a high quality design in terms of both timing and power. After that, variation-aware techniques are performed to the obtained deterministic design to improve the robustness of the design and the productivity.

B. Contribution

This dissertation heavily addresses both design stages and proposes various algorithms to address the challenge of obtaining a design with high frequency, low power and high robustness. For deterministic optimizations, new buffer insertion and gate sizing techniques are proposed. For variation-aware optimizations, new lithography-driven design techniques and post-silicon tuning-driven techniques are proposed.

1. Fast Algorithms for Slew Constrained Minimum Cost Buffering

As interconnect scales slower than device, interconnect delay has become a significant bottleneck for circuit performance [3, 4]. As an effective technique to reduce interconnect delay, buffer insertion has been widely used in practice. For example, it has been well documented [3, 5] that the number of buffers on a chip is rising dramatically. Osler [5] cites two IBM ASIC designs where 25% of the gates are buffers. In addition, interconnect resistivity causes signal integrity to degrade more quickly with advancing technologies. Thus, buffers also need to be inserted on long interconnects to meet slew constraints, even if these nets are not timing critical.

In reality, slew constraint is much more prevalent than timing constraint: it is reported in [5] that only a fraction (roughly 5-10%) of nets need to be re-buffered for delay optimization; for the remaining fraction (roughly 90-95%), the slew based buffer insertion was sufficient to meet the net's timing constraint. In other words, it is sufficient to buffer all nets to fix slew violations without worrying about delay. Those small fraction of buffered nets that subsequently show up as critical can then be re-buffered with a delay based objective function. As the sheer number of buffers can degrade overall design performance by forcing the rest of the logic to be spread further apart to accommodate those buffers, one also wishes to use as little buffering

resources as possible.

We first formulate the problem as to find the minimum area buffering solution such that slew constraints are satisfied. Based on the new formulation, the general slew buffering problem is shown to be NP-Complete. Despite the difficulty of the problem, some highly efficient and practical algorithms are proposed. First, for a single buffer type, an optimal linear time solution is achievable. Second, for multiple buffer types, a very efficient dynamic programming based optimal slew buffering algorithm is designed under the assumption that the input slew to each buffer is fixed. Experiments show that compared to slew constrained timing buffering, $> 90\times$ speedup is achieved while still saving area. Third, if the input slew to each buffer is not fixed, the dynamic programming cannot be easily applied since the upstream knowledge is needed to compute the input slew. We propose a maximum matching based new algorithm to handle this difficult case. Experimental results demonstrate that up to 21.9% buffer area can be further saved. Fourth, when buffer positions can be freely chosen, slew buffering may allow more efficient buffer usage. A continuous slew buffering algorithm incorporating adaptive buffer selection idea is proposed for this purpose. It handles 1000 nets in only 30 seconds and often extra 5% buffer area saving can be obtained. The algorithm is further extended to handle blockages which makes it ready for practical use. Refer to Chapter II for the details of the project.

2. Gate Sizing for Cell Library-Based Designs

In addition to interconnect optimizations, gate delay optimizations are studied in the dissertation. With increasing time-to-market pressure and shortening semiconductor product cycles, more and more chips are being designed with library-based methodologies. In cell library based designs, a handful set of gate sizing techniques exist. However, most of them handle the continuous gate sizing problem which is based on

the assumption that gate sizes can be any values within certain range. When gate implementations are restricted to discrete sizes, as in reality, the problem becomes much more difficult and very few approaches (see, e.g., [6]) are known. On the other hand, a large number of realistic cell libraries are "sparse". For example, when the cell sizes are geometrically spaced instead of uniformly spaced, significant sparseness is introduced. Refer to [7] for some realistic sparse libraries. Geometrically spaced gate sizes are desired because uniformly spaced gate sizes would result in a large number of gate sizes and managing this large volume of data is difficult [7]. Furthermore, it is proven in [7] that under certain conditions, the set of optimal gate sizes must satisfy the geometric progression.

In this project, we propose a novel gate sizing technique which directly handles discrete gate sizes. As many efficient solutions exist for the continuous gate sizing problem, one might think of obtaining a discrete solution through rounding a continuous solution. This is very fast but often results in large timing violations. In contrast, the method proposed by Coudert [6], which is based on the multi-dimensional descent optimization, directly handles the discrete sizes. However, it has some trial-and-error flavor and has room for further improvement. A dynamic programming approach can search solutions more systematically and thus has the potential to generate high quality solutions. However, it may suffer from large computation overhead, which imposes a great challenge to our problem.

The key idea of the new algorithm is to integrate the solution quality of dynamic programming with the short runtime of rounding continuous solution. That is, we narrow down the searching space of dynamic programming under the guidance from a best continuous solution. Thus instead of checking every implementation, our algorithm only investigates a number of discrete implementations around the best continuous solution. This enables us to find solutions with quality close to the best

continuous case and at the same time obtain huge speedup in computation. Our experimental results demonstrate that nearest rounding often leads to significant timing violations and compared to [6], our algorithm saves up to 21% area while satisfying the timing constraint. Refer to Chapter III for the details of the project.

3. Pattern Sensitive Placement for Manufacturability

Lithography-induced variation is a main source of variations. It is due to the fact that with technology scaling, demands for minimum feature sizes have outpaced the advances in lithography hardware solutions and smaller amount of variations can be tolerated, which are evident from Table I. These impose great challenges on manufacturing reliability. In current lithography technology, $193nm$ wavelength is used to print $45nm$ features. This leads to a lot of refractive effects and images on wafer have remarkable mismatches from mask layouts. Lithography-induced variation also aggravates. As more variations are presented with e.g., gate length, timing and power of circuits are significantly affected.

Currently, semiconductor industry heavily relies on *resolution enhancement techniques* (RETs) for improving printability. Roughly speaking, printability refers to the difficulty in obtaining a good match between the intended image and the printed image in lithography process. Printability is often measured by *critical dimension* (CD) accuracy, which refers to the size of thin features which are difficult to print reliably. Thus, achieving high CD accuracy means that the printed patterns well match the desired ones. RETs are effective in improving CD accuracy. However, increasingly shrinking features on the die and increasing complexity of the design over-stretch the capability of RETs. This problem aggravates when RETs are applied to the layouts which are not lithography friendly. Furthermore, RETs often complicate photomark shapes and introduce large additional cost to photomask fabrication, which makes

RETs expensive to apply. To attack the above issues, efforts are needed in all process and design stages. With respect to physical design, manufacturability-aware methodologies would be performed in order to reduce the burden of manufactures and make RETs less expensive to apply. Furthermore, since variability has big impact on power, design for manufacturability also tends to mitigate the lithography-induced variations on power.

Placement of cells has remarkable effect on printability. This is due to the fact that gate lengths for transistors on the boundary regions of a cell significantly depend on its neighboring cells. Although sound library cell design can achieve high printability for internal transistors, it cannot handle the boundary transistors. On the other hand, as the gate length keeps shrinking with technologies, the placement will affect deeper and deeper regions of the cells.

In this project, the problem of cell placement considering manufacturability is studied. Instead of designing a new cell placer, our goal is to tune any existing cell placement solution to be lithography friendly. For this purpose, three algorithms are proposed, which are cell flipping algorithm, single row optimization approach and multiple row optimization approach. These algorithms are based on dynamic programming and graph theoretic approaches, and can provide different tradeoff between critical dimension (CD) variation reduction and wirelength increase. Using lithography simulations, our experimental results on realistic netlists and cell library demonstrate that over 15% CD variation reduction can be obtained by the new approaches while only less than 1% additional wire is introduced. Refer to Chapter IV for the details of the project.

4. Unified Adaptivity Optimization of Clock and Logic Signals

In addition to lithography-driven optimizations, statistical optimization approaches are a class of effective approaches to handle variations and improve yield. In statistical optimizations, each gate or wire delay is modeled as a probabilistic density function (PDF) in contrast to a deterministic value. Based on that, the goal of the circuit optimizations is to optimize the PDF of the whole circuit delay. They are performed in the pre-silicon phase (e.g., [8, 9, 10]). That is, circuit parameters are determined in design time for yield optimization. With statistical variation models, they obtain the statistically optimized design and apply the design to all the dies. Although the optimized design is of good quality in statistical sense, the design is not necessarily ideal for each individual fabricated chip. Specific circuit parameter variations on the die cannot be mitigated. In addition, reliable statistical variation models are not easy to obtain [11].

In contrast to pre-silicon statistical optimizations, post-silicon tuning methodology can tune some circuit parameters after the chip is fabricated. This enables us to mitigate the specific circuit parameter variations on the individual chip to satisfy the design target. As a result, the timing yield can be significantly improved [12, 11].

Clearly, it is highly desirable to perform circuit adaptivity optimization for post-silicon tuning. Since making a circuit element post-silicon tunable necessarily introduces overhead, adaptivity optimization for post-silicon tuning aims to provide large tunability with small overhead. Previous works focus on either logic signal tuning (e.g., [12, 13, 11]) or clock signal tuning (e.g., [14, 15]). These approaches are effective, however, the resource utilization is not necessarily efficient since the interaction between logic circuit and clock network is not explored. Performing unified adaptivity optimization on clock and logic signals has the potential to significantly reduce

overhead while still having large tunability for achieving yield target.

Our unified optimization is based on a novel linear programming formulation which can be efficiently solved by an advanced robust linear programming technique. Due to the discrete nature of the problem, the continuous solution obtained from linear programming is then efficiently discretized. This procedure involves binary search accelerated dynamic programming, batch based optimization, and Latin Hypercube sampling based fast simulation. Our experimental results demonstrate that up to 50% area cost reduction can be obtained by the unified optimization compared to optimization on logic or clock alone. In addition, the proposed discretization approach significantly outperforms the alternatives in terms of solution quality and runtime. Refer to Chapter V for the details of the project.

CHAPTER II

FAST ALGORITHMS FOR SLEW CONSTRAINED MINIMUM COST BUFFERING

As a prevalent constraint, sharp slew rate is often required in circuit design which causes a huge demand for buffering resources. This problem requires ultra-fast buffering techniques to handle large volume of nets, while also minimizing buffering cost. This problem is intensively studied in this paper. First, a highly efficient algorithm based on dynamic programming is proposed to optimally solve slew buffering with discrete buffer locations. Second, a new algorithm using the maximum matching technique is developed to handle the difficult cases in which no assumption is made on buffer input slew. Third, an adaptive buffer selection approach is proposed to efficiently handle slew buffering with continuous buffer locations. Fourth, buffer blockage avoidance is handled, which makes the algorithms ready for practical use.

Experiments on industrial netlists demonstrate that our algorithms are very effective and highly efficient: we achieve about 90× speed up and save up to 20% buffer area over the commonly-used van Ginneken style buffering. The new algorithms also significantly outperform previous works that indirectly address the slew buffering problem[1].

A. Introduction

As VLSI technology moves to the 65 nm node and beyond, it has been well documented [3, 5] that the number of buffers on a chip is rising dramatically. Osler [5]

[1]Copyright ©2007 IEEE. Reprinted, with permission, from S. Hu, C. J. Alpert, J. Hu, S. Karandikar, Z. Li, W. Shi and C. N. Sze, Fast algorithms for slew constrained minimum cost buffering, IEEE Transactions on Computer-Aided Design, Vol. 26, No. 11, pp. 2009-2022, November, 2007.

cites two IBM ASIC designs where one-fourth of the gates are buffers. For some multi-million gate ASICs, more than a million buffers are required today. This is a surprise to no one as devices continue to scale more quickly than interconnects. Higher relative interconnect resistance forces buffers to be placed closer together to achieve optimal performance. In addition, interconnect resistivity also causes signal integrity to degrade more quickly with each advancing technology. Thus, buffers need to be inserted on long interconnects to meet slew constraints, even if these nets are not timing critical.

In reality, slew constraint is much more prevalent than timing constraint: it is reported in [5] that only a fraction (roughly 5-10%) of nets need to be re-buffered for delay optimization; for the remaining fraction (roughly 90-95%), the slew based buffer insertion was sufficient to meet the net's timing constraint. In other words, it is sufficient to buffer all nets to fix slew violations without worrying about delay. Those small fraction of buffered nets that subsequently show up as critical can then be re-buffered with a delay based objective function. In the IBM physical synthesis methodology [5], buffers are inserted for satisfying slew constraints early, so that timing analysis uses legal slew constraints. Later, buffers on critical nets are ripped up and re-buffered for delay.

The sheer number of buffers can degrade overall design performance by forcing the rest of the logic to be spread further apart to accommodate those buffers. The buffers themselves are a drain on power and can cause other gates to be sized to higher power levels since they are now further apart on the chip. Therefore, a significant part of the performance of the design depends on using as little buffering resources as possible. van Ginneken's algorithm [16] and its derivative extensions [17, 18, 19, 20, 21, 22] are very effective for delay optimization. Further, Lillis' data structure [17] allows trading off delay for cost to more efficiently use buffer resources, yet this is

still suboptimal for area.

From a practical point of view, slew buffering should be as important as timing driven buffering. Unfortunately, there is very little previous work on it. For related works that consider slew and/or noise constraints [17, 23, 18, 24], they still optimize for delay instead of handling these constraints separately. Buffering of non-critical nets using these techniques may result in unnecessary runtime and resource overhead. Note that the work of [25] also addresses slew constraints without regards to delay. However, that work does not actually model slew; it simplifies the slew constraint to be equivalent to a capacitance constraint which means that interconnect resistivity is not modelled. While appropriate for very large fanout nets (e.g., over 1000 sinks), it essentially becomes equivalent to length-based buffering [26]. Length-based buffering [26] tries to achieve a similar result of slew buffering in spirit. However, we show that it can be area inefficient especially in handling multi-fanout nets.

This work proposes a new buffering formulation: find the minimum area (or cost) buffering solution such that slew constraints are satisfied. In this formulation, one does not need to know required arrival time at sinks, so it can be used earlier in the design flow than traditional buffering. It can be done totally independently of timing analysis, i.e., incremental timing is not required between buffering of individual nets. Based on the new formulation, the general slew buffering problem is shown to be NP-Complete. Despite the difficulty of the problem, some highly efficient and practical algorithms are proposed in this paper:

1. For a single buffer type, an optimal linear time solution is achievable by greedy algorithm under the assumption that the input slew to each buffer is fixed.

2. For multiple buffer types, a very efficient optimal slew buffering algorithm is designed under the assumption that the input slew to each buffer is fixed. Ex-

periments show that compared to slew constrained timing buffering, about 90× speedup is achieved while still saving area.

3. If the input slew to each buffer is not fixed, the dynamic programming cannot be easily applied since the upstream knowledge is needed to compute the input slew. We propose a maximum matching based new algorithm to handle this difficult case. Experimental results demonstrate that up to 21.9% buffer area can be further saved.

4. When buffer positions can be freely chosen, slew buffering may allow more efficient buffer usage. A continuous slew buffering algorithm incorporating adaptive buffer selection idea is proposed for this purpose. It handles 1000 nets in only 30 seconds and often extra 5% buffer area saving can be obtained.

5. Buffering with blockage is handled in this paper, which makes the algorithms ready for practical use.

Although there is a close relationship between slew buffering and timing buffering, the two buffering algorithms are actually very different. For example, in slew buffering, inserting one buffer may only generate one new non-dominated solution. However, in timing buffering, numerous new non-dominated solutions can be introduced. Refer to Section b for details.

The rest of the chapter is organized as follows: Section B formulates the slew buffering problem. Section C presents the NP-Completeness proof for the general slew buffering problem. Section D describes the proposed slew buffering algorithms. Section E describes two related buffering algorithms for comparison. Section E presents the experimental results with analysis. A summary of work is given in Section G.

B. Preliminaries

The input to the slew buffering problem includes a routing tree $T = (V, E)$, where $V = \{s_0\} \cup V_s \cup V_n$, and $E \subseteq V \times V$. For simplicity, the routing tree is assumed to be a binary tree in this paper. Trees in other topologies can be converted to a binary tree (see, e.g., [20]). Vertex s_0 is the *source* vertex, V_s is the set of *sink* vertices and V_n is the set of *internal* vertices. Each sink vertex $s \in V_s$ is associated with sink capacitance C_s. Each edge $e \in E$ is associated with lumped resistance R_e and capacitance C_e. A buffer library B contains different types of buffers. Each type of buffer b has a cost W_b, which can be measured by area or any other metric, depending on the optimization objective. Without loss of generality, we assume that the driver at source s_0 is also in B. A function $f : V_n \rightarrow 2^B$ specifies the types of buffers allowed at each internal vertex. That is, for each vertex v, $f(v)$, which is a subset of 2^B, specifies the buffer types allowed at v.

The *slew rate* of a signal refers to the rising or falling time of a signal switching. A commonly used definition of slew is the 10/90 slew and it is adopted in this paper, where 10/90 slew refers to the time difference between when the waveform crosses the 90% point and the 10% point. Some other definitions, such as 20/80 or 30/70 slew, are also used in practice when the waveform has slowly rising or falling tail. The slew model employed in this work is chosen for its simplicity and is essentially equivalent to the Elmore model for delay. More accurate wire and gate delay models may be used if more accuracy is desired. Given that the motivation for the proposed buffering formulation lies in the requirement to efficiently buffer a large number of nets, this slew model is appropriate.

The slew model can be explained using a generic example which is a path p from node v_i (upstream) to v_j (downstream) in a buffered tree. There is a buffer (or the

driver) b_u at v_i, and there is no buffer between v_i and v_j. The slew rate $S(v_j)$ at v_j depends on both the output slew $S_{b_u,out}(v_i)$ at buffer b_u and the slew degradation $S_w(p)$ along path p (or wire slew), and is given by [27]:

$$S(v_j) = \sqrt{S_{b_u,out}(v_i)^2 + S_w(p)^2}. \tag{2.1}$$

The slew degradation $S_w(p)$ can be computed with Bakoglu's metric [28] as

$$S_w(p) = \ln 9 \cdot D(p), \tag{2.2}$$

where $D(p)$ is the Elmore delay from v_i to v_j.

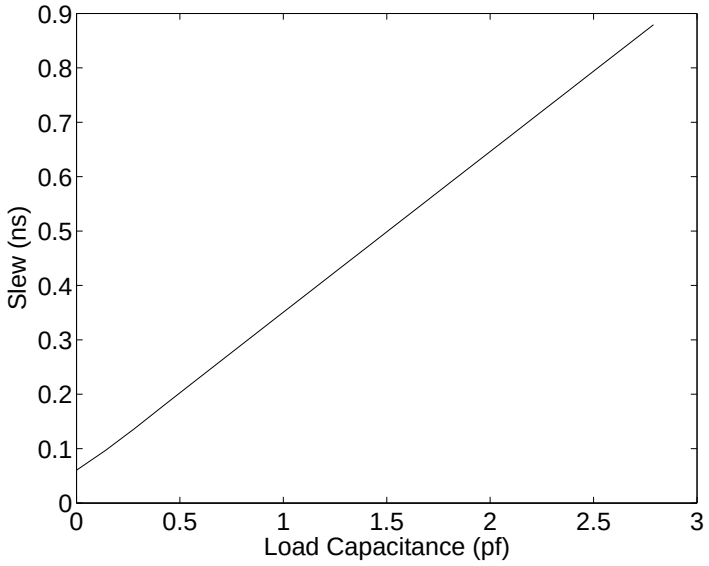


Fig. 2. The slew-capacitance curve of an inverter.

The output slew of a buffer, such as b_u at v_i, depends on the input slew at this buffer and the load capacitance seen from the output of the buffer. Usually, the dependence is described as a 2-D lookup table. In addition to handling the general case of arbitrary input slew, our work includes fast algorithms assuming a fixed input slew which is normally a conservative estimation (the slew constraint).

This assumption allows us to process large volume of nets quickly with small solution degradation. For fixed input slew, the output slew of buffer b at vertex v is then given by

$$S_{b,out}(v) = R_b \cdot C(v) + K_b, \tag{2.3}$$

where $C(v)$ is the downstream capacitance at v, R_b and K_b are empirical fitting parameters. This is similar to empirically derived K-factor equations [29]. We call R_b the slew resistance and K_b the intrinsic slew of buffer b. Figure 2 shows a slew curve of one inverter generated by EinsTimer [30]. The linear order model is quite reasonable as seen from Figure 2.

A buffer assignment γ is a mapping $\gamma : V_n \rightarrow B \cup \{\overline{b}\}$ where $\overline{b}$ denotes that no buffer is inserted. The cost of a solution γ is $W(\gamma) = \sum_{b \in \gamma} W_b$. With the above notations, the basic slew buffering problem can be formulated as follows.

Discrete Slew Constrained Minimum Cost Buffer Insertion Problem: Given a binary routing tree $T = (V, E)$, possible buffer positions, and a buffer library B, to compute a buffer assignment γ such that the total cost $W(\gamma)$ is minimized such that the input slew at each buffer or sink is no greater than a constant α.

Note that the continuous slew buffering problem is also considered in this paper where buffer positions can be freely chosen in a routing tree. A first glance at the above closed form model might suggest close relationship between timing buffering and slew buffering, however, they actually significantly differ. A detailed analysis is presented in Section b. Before closing this section, we note the following computational complexity result:

Theorem 1: The minimum cost slew buffering problem is NP-Complete, if the size of the buffer library is not constant and the cost of each buffer can be an arbitrary integer.

Refer to Section C for the proof. Since the size of the buffer library is bounded and the buffer area is not an arbitrary value in reality, our algorithms perform very well in practice.

C. Complexity of Slew Buffering Problem

Proof of Theorem 1:

The problem is clearly in NP. We reduce from the minimum cost timing buffering problem with unbounded buffer library size[1] to show that the minimum cost slew buffering problem with unbounded buffer library size is NP-Complete. Let Q, R, C, W denote the required arrival time (RAT), resistance, capacitance and cost, respectively. It is shown in [19] that computing a timing buffering for the tree in Figure 3 with RAT at driver $Q_{s_0} \geq 0$ and the total buffer cost at most $M = N + \sum_{i=1}^{n} N^i$ is NP-Complete. Driver resistance is set to $R_{s_0} = N^n$, sink capacitance and sink RAT are listed in Table II, and the buffer library information is shown in Table III, where N is a sufficiently large positive integer, $x_1, x_2, \ldots, x_{2n}$ are positive integers such that $\sum_{1}^{2n} x_i = 2N$, and there are n sinks and $2n$ buffer types in the buffer library.

We set intrinsic slew and intrinsic delay to zero, and slew resistance equal to driving resistance for each buffer type and driver. Furthermore, every edge in the tree has zero wire capacitance and zero wire resistance. It is then easy to check that the slew rate is equal to the delay in value. For example, delay and slew rate at v_1 are both $R_{s_0} \cdot C_{b_1}$ assuming that b_1 is placed at v_1.

Given an instance of minimum cost timing buffering problem, we construct an instance of minimum cost slew buffering problem as follows. We reuse the routing tree in Figure 3 except that each sink s_i is changed to a sink s'_i, where $C_{s'_i} = \frac{C_{s_i}}{N}$. A

[1]That is, the number of buffer types is not constant.

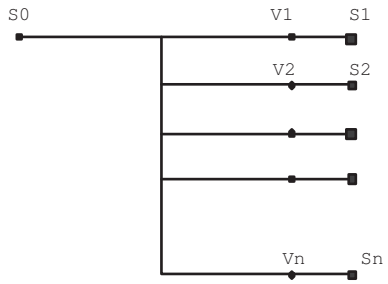


Fig. 3. Underlying routing tree and buffer positions [19].

Table II. C, Q values for sinks [19].

Sink s_i	Sink capacitance C_{s_i}	Sink RAT Q_{s_i}
s_1	N^{n+2}	$N^{n+1} + N^{n+2}$
s_2	N^{n+1}	$N^{n+1} + N^{n+2}$
...	...	...
s_n	N^3	$N^{n+1} + N^{n+2}$

critical fact used in [19] is that every buffer position v_i must be inserted with a buffer, and this buffer must be either b_{2i-1} or b_{2i}.

We claim that there is a solution for the instance of slew buffering problem with slew constraint $\alpha = N^{n+1}$ and the total buffer cost at most M if and only if there is a solution for the instance of minimum cost timing buffering problem with $Q_{s_0} \geq 0$ and with the same cost bound.

We begin with the "only if" direction. Since slew constraint is set to N^{n+1}, it follows that the delay between s_0 and v_i and delay between v_i and s_i' is no more than N^{n+1} each. Since $C_{s_i'} = \frac{C_{s_i}}{N}$, delay between v_i and s_i is bounded above by N^{n+2}. Noting that $Q_{s_i} = N^{n+2} + N^{n+1}$, we have $Q_{s_0} \geq 0$.

For the "if" direction, since we must insert one of b_{2i-1} and b_{2i} at every v_i, delay

Table III. C, R, W values for each buffer type [19].

Buffer b_i	Driving resistance R_{b_i}	Input capacitance C_{b_i}	Cost (Area) W_{b_i}
b_1	1	x_1	$x_2 + N^n$
b_2	1	x_2	$x_1 + N^n$
b_3	N	x_3	$x_4 + N^{n-1}$
b_4	N	x_4	$x_3 + N^{n-1}$
...	...	...	...
b_{2n-1}	N^{n-1}	x_{2n-1}	$x_{2n} + N$
b_{2n}	N^{n-1}	x_{2n}	$x_{2n-1} + N$

between v_i and s_i is N^{n+2}, and thus the slew rate at s_i' is N^{n+1}. Since total delay from s_0 to any sink s_i is no larger than $N^{n+1} + N^{n+2}$, one sees that delay between s_0 and any v_i is bounded above by N^{n+1}. Therefore, in the buffered tree, slew rate at any buffer position/sink is bounded above by α, which completes the proof.

D. Slew Constrained Minimum Cost Buffering Algorithms

1. Overview of Classic Timing-Driven Buffering

To understand the context of the presented algorithms and to define notation, this section begins with a brief overview of van Ginneken/Lillis [16, 17] algorithm. The algorithm proceeds bottom-up from the leaf nodes toward the driver along a given routing tree. A set of candidate solutions is kept updated during the process. Each solution is associated with a three-tuple (C, W, Q), where C denotes the downstream capacitance at the current node, W denotes the cost (i.e., area) of the solution and

Q refers to the required arrival time (RAT).

Suppose that a solution γ_v at position v must "propagate" to an upstream position u and there is no branching point in between. If no buffer is placed at u, then only wire delay needs to be considered. Therefore, the new solution γ_u can be computed as

$$\begin{aligned} C(\gamma_u) &= C(\gamma_v) + C_e, \\ W(\gamma_u) &= W(\gamma_v), \\ Q(\gamma_u) &= Q(\gamma_v) - D_e, \end{aligned} \tag{2.4}$$

where $e = (u, v)$ and $D_e = R_e(\frac{C_e}{2} + C(\gamma_v))$. Otherwise, suppose that we add a buffer b_i at u. γ_u can be then computed as

$$\begin{aligned} C(\gamma_u) &= C_{b_i}, \\ W(\gamma_u) &= W(\gamma_v) + W_{b_i}, \\ Q(\gamma_u) &= Q(\gamma_v) - D_{b_i} - D_e \end{aligned} \tag{2.5}$$

after buffer insertion. In Eqn. (2.5), D_{b_i} refers to the buffer delay and is computed as $D_{b_i} = R'_{b_i} \cdot C(u) + K'_{b_i}$, where R'_{b_i} is the driving resistance of b_i but not the slew resistance of b_i, and K'_{b_i} is the intrinsic buffer delay.

An important concept in van Ginneken/Lillis algorithm are *non-dominated solutions*. For any two solutions γ_1, γ_2 *at the same node*, γ_1 *dominates* γ_2 if $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$ and $Q(\gamma_1) \geq Q(\gamma_2)$. Whenever a solution becomes dominated, it is removed from the solution set. Therefore, only solutions excel in at least one aspect of downstream capacitance, buffer cost and RAT can survive.

For handling branch merging, suppose that we have obtained all the non-dominated solutions of left branch T_l and right branch T_r at a branching point v_t[1]. Denote the

[1]For two branches, we arbitrarily assign them to be left branch and right branch.

left-branch solution set and the right-branch solution set by Γ_l and Γ_r, respectively. The merging process is performed as follows. For each solution $\gamma_l \in \Gamma_l$ and each solution $\gamma_r \in \Gamma_r$, generate a new solution γ' according to:

$$\begin{aligned} C(\gamma') &= C(\gamma_l) + C(\gamma_r), \\ W(\gamma') &= W(\gamma_l) + W(\gamma_r), \\ Q(\gamma') &= \min\{Q(\gamma_l), Q(\gamma_r)\}. \end{aligned} \tag{2.6}$$

At a high level, van Ginneken/Lillis algorithm builds the solution set in a bottom-up fashion. Assume that we have computed all feasible non-dominated solutions at a buffer position v. For the immediately upstream buffer position u (without passing any branching point), we first propagate all solutions up there through performing wire insertion of (u, v) to each solution. The propagated solutions resemble the choices when no buffer is inserted at u. Subsequently, for each propagated solution, we compute a new solution for inserting each buffer. The new solution is inserted into the solution set as long as it is not dominated by any existing one. The solution set is meanwhile updated to prune the solutions being dominated by the newcomer. At a merging point, we carry out the process just described to generate the new solution set. In this way, we keep climbing up the routing tree until the driver is met. After pruning solutions violating the timing constraint at driver, we select the best solution as the one with the smallest cost.

2. Discrete Slew Buffering Assuming Fixed Input Slew

a. Algorithm

Our algorithms share the same dynamic programming framework as timing buffering [16, 17] in appearance, but have critical underlying differences which will be analyzed in Section b and Section c.

In the dynamic programming framework, a set of candidate solutions are propagated from the sinks toward the source along the given tree. Each solution γ is characterized by a three-tuple (C, W, S), where C denotes the downstream capacitance at the current node, W denotes the cost of the solution and S is the accumulated slew degradation S_w defined in Eqn. (2.2). At a sink node, the corresponding solution has C equal to the sink capacitance, $W = 0$ and $S = 0$. The solution propagation is accomplished by the following operations.

Consider to propagate solutions from a node v to its parent node u through edge $e = (u, v)$. A solution γ_v at v becomes solution γ_u at u, which can be computed as $C(\gamma_u) = C(\gamma_v) + C_e, W(\gamma_u) = W(\gamma_v)$ and $S(\gamma_u) = S(\gamma_v) + \ln 9 \cdot D_e$ where $D_e = R_e(\frac{C_e}{2} + C(\gamma_v))$.

In addition to keeping the unbuffered solution γ_u, a buffer b_i can be inserted at u to generate a buffered solution $\gamma_{u,buf}$ which can be then computed as $C(\gamma_{u,buf}) = C_{b_i}, W(\gamma_{u,buf}) = W(\gamma_v) + W_{b_i}$ and $S(\gamma_{u,buf}) = 0$.

When two sets of solutions are propagated through left child branch and right child branch to reach a branching node, they are merged. Denote the left-branch solution set and the right-branch solution set by Γ_l and Γ_r, respectively. For each solution $\gamma_l \in \Gamma_l$ and each solution $\gamma_r \in \Gamma_r$, the corresponding merged solution γ' can be obtained according to: $C(\gamma') = C(\gamma_l) + C(\gamma_r), W(\gamma') = W(\gamma_l) + W(\gamma_r)$ and $S(\gamma') = \max\{S(\gamma_l), S(\gamma_r)\}$. To ensure that the worst case in the two branches still satisfies slew constraint, we take the maximum slew degradation for the merged solution.

For any two solutions γ_1, γ_2 *at the same node*, γ_1 *dominates* γ_2 if $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$ and $S(\gamma_1) \leq S(\gamma_2)$. Whenever a solution becomes dominated, it is pruned from the solution set without further propagation. A solution γ can also be pruned when it is infeasible, i.e., either its accumulated slew degradation $S(\gamma)$ or the

slew rate of any downstream buffer in γ is greater than the slew constraint α.

b. Critical Differences from Timing Buffering

When a buffer b_i is inserted into a solution γ, $S(\gamma)$ is set to zero and $C(\gamma)$ is set to $C(b_i)$. This means that inserting one buffer may bring *only one new solution*, namely, the one with the smallest area, W. However, in minimum cost timing buffering, a buffer insertion may result in many non-dominated (C, W, Q) tuples with the same C value, where Q denotes the require arrival time.

Consequently, in slew buffering, at each buffer position *along a single branch*, at most $|B|$ new solutions can be generated through buffer insertion since C, S are the same after inserting each buffer. In contrast, buffer insertion in the same situation may introduce many new solutions in timing buffering. This sheds light on why slew buffering can be much more efficiently computed.

Another important fact is that the slew constraint is in some sense close to length constraint. In slew buffering, solutions can soon become infeasible if we do not add a buffer into it and thus many solutions, which are only propagated through wire insertion, are often removed soon. An extreme case demonstrating this point is that in standard timing buffering, the solutions with no buffer inserted can always live until being pruned by driver given a loose timing constraint. This may not happen in slew buffering: this kind of solutions soon become infeasible as long as the slew constraint is not too loose.

Due to these special characteristics of the slew buffering problem, a *linear time* optimal algorithm for buffering with a single buffer type is possible. In timing buffering, it is not known how to design a *polynomial time* algorithm in this case. Refer to Section D for the details. From these facts, the basic differences between these two somewhat related buffering problems are clear.

c. Implementation Experiences

This section presents a fast algorithm for the slew buffering problem. Except for special efforts for handling S, the new algorithm works as [17]. Refer to Figure 4 for the pseudocode of the proposed algorithm. For consistency, we insert a dummy buffer b_0 to a position when no buffer is to be inserted there.

The bolded part in Figure 4 shows the difference between the slew buffering algorithm and [17]. First, S, which represents accumulated slew degradation on wire, is a newly introduced term and thus does not exist in van Ginneken/Lillis' algorithm. Second, the SolutionSetUpdate procedure shown in Figure 5 significantly differs: a new solution is first checked for feasibility; if the slew constraint is satisfied, the domination check/elimination procedure for the solution set will be carried out.

We are to elaborate some implementation details in domination check as well as domination elimination. In the algorithm, the solution set is stored using a linked list where elements are in no particular order. The straightforward linear search is carried out into the solution list by each newcomer for domination checking and meanwhile, the solution list is updated for domination elimination. This simple implementation gives excellent performance due to the critical fact that size of solution set here is always small. We usually have less than 20 non-dominated solutions at driver in each routing tree, and the typical total runtime over 1000 nets is less than 20 seconds. Refer to Section E for the details.

Therefore, in contrast to using range search tree to prune the dominated solutions as in [17], the simple linked list implementation works very well here. We believe that the simplicity of implementation for slew buffering with fixed buffer input slew will make it widely used in practice.

One would wonder the effect of introducing the range search tree into the slew

Algorithm: Slew buffering w/ fixed input slew.

Input: T: routing tree, B: buffer library, α: slew constraint
Output: minimum cost buffer assignment γ satisfying α

1. for each sink s, build a solution set $\{\gamma_s\}$, where $W(\gamma_s) = 0$, $\boldsymbol{S(\gamma_s) = 0}$, and $C(\gamma_s) = C_s$
2. for each branching point/driver v_t in the order given by a postorder traversal of T, let T' be the two branches T_1, T_2 of v_t and Γ' be the solution set corresponding to T', do
3. for each wire e in T', in a bottom-up order, do
4. for each $\gamma \in \Gamma'$ corresponding to T', do
5. $C(\gamma) = C(\gamma) + C_e$
6. **set** $\boldsymbol{S(\gamma) = S(\gamma) + \ln 9 \cdot D_e}$
7. **SolutionSetUpdate** $\boldsymbol{(\gamma, \Gamma', b_0, \alpha)}$
8. if the current position allows buffer insertion, then
9. for each buffer type $b_i \in B$, do
10. for each $\gamma \in \Gamma'$, generate a new solution γ'
11. set $C(\gamma') = C_{b_i}$
12. set $W(\gamma') = W(\gamma') + W_{b_i}$
13. **set** $\boldsymbol{S(\gamma') = 0}$
14. **SolutionSetUpdate** $\boldsymbol{(\gamma', \Gamma', b_i, \alpha)}$
15. // merge Γ_1 and Γ_2 to Γ_{v_t}
16. set $\Gamma_{v_t} = \emptyset$
17. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate a new solution γ'
18. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
19. set $W(\gamma') = W(\gamma_1) + W(\gamma_2)$
20. **set** $\boldsymbol{S(\gamma') = \max\{S(\gamma_1), S(\gamma_2)\}}$
21. **SolutionSetUpdate** $\boldsymbol{(\gamma', \Gamma_{v_t}, b_0, \alpha)}$
22. eliminate infeasible solutions at driver and return γ with the smallest cost

Fig. 4. Slew constrained minimum cost buffering algorithm with fixed buffer input slew.

Procedure: SolutionSetUpdate w/ fixed input slew
Input: γ': a candidate solution, Γ: a solution set, b: a buffer type, α: a slew constraint **Output:** an updated solution set Γ
1. // check whether γ' violates the slew constraint 2. if $b = b_0$, then 3. return Γ if $S(\gamma') > \alpha$ 4. else 5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$ 6. // domination check and domination elimination 7. for each solution $\gamma \in \Gamma$, do 8. if $C(\gamma) \leq C(\gamma')$, $W(\gamma) \leq W(\gamma')$ and $S(\gamma) \leq S(\gamma')$, 9. return Γ 10. if $C(\gamma') \leq C(\gamma)$, $W(\gamma') \leq W(\gamma)$ and $S(\gamma') \leq S(\gamma)$, 11. remove γ from Γ 12. insert γ' into Γ and return Γ

Fig. 5. Procedure of updating solution set for slew buffering with fixed buffer input slew.

buffering algorithm. As such, the slew buffering algorithm combined with range search tree pruning [17] is also tested. Unfortunately, the slew buffering algorithm is slowed down. This phenomenon is due to the considerable amount of inherent overhead in maintaining the balanced binary search tree through e.g., rotation for each insertion/deletion in the data structure. Refer to Section E for the details.

Recall that at each buffer position, we introduce $|B|$ new solutions by buffer insertion. Thus, a branch having n buffer positions will introduce at most $n|B|$ *new* solutions. Consider to merge two branches each of which has $n_1|B|$ and $n_2|B|$ solutions, respectively, where n_1 and n_2 denote the number of buffer positions in each branch. After merging, the number of solutions is bounded by $n_1 n_2 |B|^2$. Suppose that another branch merging produces $n_3 n_4 |B|^2$ solutions. Further suppose that these resulting solutions are merged and $n_1 n_2 n_3 n_4 |B|^4$ solutions are obtained. Let n denote the number of buffer positions, m denote the number of sinks, and n_i denote the number of buffer positions at branch i. By the above process, one can see that

the total number of solutions is bounded by $(\frac{n}{m})^m|B|^m$, since $n_1 n_2 \cdots n_m \leq (\frac{n}{m})^m$ and $n_1 + n_2 + \ldots + n_m = n$.

Since we can have at most $O((\frac{n|B|}{m})^m)$ solutions at any position, a domination check at a position is performed through a traversal of the linked list consisting of $O((\frac{n|B|}{m})^m)$ solutions and thus needs $O((\frac{n|B|}{m})^m)$ time per traversal. At a buffer position, $|B|$ new solutions are introduced due to buffer insertions. The domination checks need $|B|$ traversals of the solution set, which takes $O(|B|(\frac{n|B|}{m})^m)$ time. The most time-consuming step is branch merging where we at most perform $O((\frac{n|B|}{m})^m)$ domination checks since it is the upper bound for the number of solutions at any position. Thus, a branch merging needs $O(((\frac{n|B|}{m})^m)^2)$ time. Since there are n buffer positions, the proposed algorithm returns the optimal solution in $O(n|B| \cdot (\frac{n|B|}{m})^{2m})$ time. Theoretically (though impractically), when $|B| = \Theta(n), m = \Theta(n)$, the algorithm will run in exponential time. This surprises no one given the NP-Completeness nature of the slew buffering problem.

3. Discrete Buffering without Input Slew Assumptions

a. Basic Modifications

In Section a, the output slew of a buffer (computed by Eqn. (2.3)) does not depend on the input slew. This is valid since slew resistance R_{b_i} is obtained by assuming the input slew for each buffer to be fixed at the slew constraint. Certainly, improvement in buffer area is desired if this assumption is eliminated. As such, a more complicated dynamic programming algorithm which handles *non-fixed input slew* is proposed as follows.

Our idea is to approximate continuous-valued input slew by different small-sized slew bins. That is, the input slew at each buffer position is discretized into different

input slew bins, each of which covers a range of slew rate. Clearly, better results can be obtained with finer input slew bins. Denote by l the number of input slew bins.

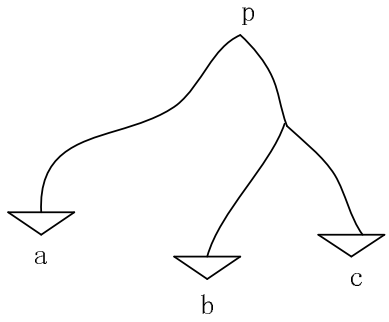


Fig. 6. An example of handling non-fixed input slew.

Suppose that a buffer is to be inserted at position p and there are three *immediate downstream* buffers in a solution γ as shown in Figure 6. As the result upstream from p is not yet known, the input slew to the buffer can be in any slew bin.

As such, in addition to C, W, S, each solution is augmented with new tuples L, U, which specify the lower bound and upper bound of the input slew to these immediate downstream buffers, respectively. In other words, the input slew is required to fall in $[L, U)$. Suppose that viewing at p, we have $n(\gamma)$ immediate downstream buffers, each of which is associated with a lower bound L_i and an upper bound U_i. Accordingly, there is an S_i representing accumulated slew degradation viewing at each immediate downstream buffer. For example, a solution at p in Figure 6 has (S_1, L_1, U_1) for the buffer inserted at a, (S_2, L_2, U_2) for b, and (S_3, L_3, U_3) for c. Therefore, each solution is characterized by (C, W, S_i, L_i, U_i), $1 \leq i \leq n$ if the solution has n immediate downstream buffers.

When a buffer is inserted at p, *at most* l new solutions are generated. They are with the same C, W, S values but with different L, U values. We say "at most" since whether a buffer with a certain input slew bin can be inserted at p needs to be

validated. For a buffer b to be inserted with the input slew bin g, denote by $[\underline{S_g}, \overline{S_g})$ the slew range of g. The buffer insertion is valid if for each immediate downstream buffer i (viewing at p, $1 \leq i \leq n(\gamma)$) in γ,

$$L_i(\gamma) \leq \sqrt{S_{b,out}(p, g, C(\gamma))^2 + S_i(\gamma)^2} \leq U_i(\gamma), \tag{2.7}$$

where $S_{b,out}(p, g, C(\gamma))$ is the output slew of the buffer b at p with g as its input slew bin and $C(\gamma)$ as its downstream capacitance, and a lookup table is used to obtain its value. Upon validation, the buffer b is inserted to γ, the number of immediate downstream $n(\gamma)$ is set to one, $S_1(\gamma)$ is set to zero, and $L_1(\gamma) = \underline{S_g}$ and $U_1(\gamma) = \overline{S_g}$.

It is often valid for a buffer with numerous input slew bins to be inserted to the same solution γ. For efficiency reason, those new solutions are merged after buffer insertion. That is, after buffer insertion, two solutions γ_1 and γ_2 are merged to form γ' if $C(\gamma_1) = C(\gamma_2), W(\gamma_1) = W(\gamma_2)$ and $U_1(\gamma_1) = L_1(\gamma_2)$, where C, W, S of γ' remain unchanged while $L_1(\gamma') = L_1(\gamma_1)$ and $U_1(\gamma') = U_1(\gamma_2)$.

Note that in branch merging, the parameter values (S, L, U) of all immediate downstream buffers for a left-branch solution γ_1 and a right-branch solution γ_2 are stored together and $n(\gamma') = n(\gamma_1) + n(\gamma_2)$.

b. Reduction to Maximum Bipartite Matching

The definition of domination needs to be accordingly modified. For two solutions with the same number of immediate downstream buffers, domination is defined *solely* on C, W, S_i, L_i, U_i. In particular, the i-th buffer in γ_1 and that in γ_2 may refer to different immediate downstream buffers. This allows a fairly effective solution pruning procedure.

Given two solutions γ_1 and γ_2, we are to decide whether there is a pairing of immediate downstream buffers of γ_1 and γ_2, respectively, such that $S_{\pi_1(j)}(\gamma_1) \leq S_{\pi_2(j)}(\gamma_2)$,

$L_{\pi_1(j)}(\gamma_1) \leq L_{\pi_2(j)}(\gamma_2)$ and $U_{\pi_1(j)}(\gamma_1) \geq U_{\pi_2(j)}(\gamma_2)$ for each pair j where $1 \leq j \leq n(\gamma_1) = n(\gamma_2)$, and $\pi(\cdot)$ denotes the permutation of indices of immediate downstream buffers. If this is the case, together with $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$, we conclude that γ_1 dominates γ_2.

An example would be helpful to illustrate the above definition. Assume that γ_1, γ_2 both have three immediate downstream buffers. Suppose that (S_i, L_i, U_i) for γ_1 are $(3, 10, 60), (5, 30, 65), (3, 20, 50)$, and for γ_2 are $(5, 25, 35), (6, 50, 55), (10, 15, 35)$. γ_1 dominates γ_2 on (S, L, U) since $(3, 10, 60)$ dominates $(10, 15, 35)$, $(5, 30, 65)$ dominates $(6, 50, 55)$, and $(3, 20, 50)$ dominates $(5, 25, 35)$.

Given two solutions, we need to answer whether such pairing exists. The straightforward computation is inefficient since L, U may heavily overlap. As such, we reduce it to the maximum bipartite matching problem for an efficient solution. To check whether γ_1 dominates γ_2, for each $(S_i(\gamma_1), L_i(\gamma_1), U_i(\gamma_1))$ in γ_1, a set of tuples, denoted by $\psi_i(\gamma_1)$, consisting of all $(S_j(\gamma_2), L_j(\gamma_2), U_j(\gamma_2))$ in γ_2 is computed such that the former three-tuple dominates each of the latter three-tuples. A graph $G = (V, E)$ is constructed as follows. Represent each three-tuple by a vertex. A vertex corresponding to the i-th tuple in γ_1 links to the vertices corresponding to $\psi_i(\gamma_1)$. A bipartite graph is formed in this way since there are no links between nodes representing tuples in the same solution. For these two groups of vertices, the task is to answer whether there is a node-wise pairing (each from different groups) of cardinality $n(\gamma_1)$.

This is a maximum matching problem, which is to compute an edge set E' of maximum cardinality from E such that each vertex in V is incident to at most one edge of E'. Domination (on S, L, U) follows if E' is of cardinality of $n(\gamma_1)$. The best bipartite matching algorithm runs in $O(\sqrt{|V|}|E| \log(|V|^2/|E|)/\log|V|)$ time [31]. In this paper, an efficient practical implementation [32] based on the scaling push-relabel approach, is adopted. Refer to Figure 7 and Figure 8 for the algorithms for slew

buffering without fixed input slew assumption.

We are to present the complexity analysis of the slew buffering algorithm with non-fixed input slew. Instead of $|B|$ new solutions in the fixed-input slew case, at most $l|B|$ new solutions can be generated at each buffer position due to buffer insertions in slew buffering algorithm with non-fixed input slew. Thus, the total number of solutions is always bounded above by $O((\frac{nl|B|}{m})^m)$ where n is the number of buffer positions and m is the number of sinks. The complexity analysis goes the same as in Section c except that one additional step, which is due to using maximum bipartite matching in domination check, needs to be considered. A solution can have at most m (S, L, U) tuples since there are only m sinks. Therefore, in our maximum matching problem, $|V| \leq m$ and $|E| \leq m^2$. Plugging these numbers into $O(\sqrt{|V|}|E| \log(|V|^2/|E|)/\log|V|)$, we have that the bipartite matching algorithm in [31] runs in $O(m^{2.5})$ time. A domination check needs to perform a traversal of the solution set and for each traversed solution, the above $O(m^{2.5})$ algorithm is carried out. Thus, a domination check needs $O((\frac{nl|B|}{m})^m \cdot m^{2.5})$ time. As in Section c, the most time-consuming step is branch merging where $O((\frac{nl|B|}{m})^m)$ domination checks may happen. It is then easy to see that the total runtime is bounded above by $O(nl|B| \cdot (\frac{nl|B|}{m})^{2m} \cdot m^{2.5})$.

4. Continuous Slew Buffering

What we have considered so far is the discrete slew buffering problem. It is expected that the total buffer area can be reduced if buffer positions are freely chosen in the routing tree. The following continuous slew buffering algorithm settles this problem. We begin with a simple case:

Theorem 2: For a single buffer type, the optimal slew buffering can be computed in linear time under the assumption that the input slew to each buffer is fixed.

Algorithm: Slew buffering w/ non-fixed input slew.

Input: T, B, α, input slew bins
Output: minimum cost buffer assignment γ satisfying α

1. build solutions at each sink for each input slew bin
2. for each branching point/driver v_t in the order given by a postorder traversal of T, let T' be two branches T_1, T_2 of v_t and Γ' be the solution set, do
3. for each wire e in T', in a bottom-up order, do
4. for each $\gamma \in \Gamma'$ corresponding to T', do
5. $C(\gamma) = C(\gamma) + C_e$
6. set $S(\gamma) = S(\gamma) + \ln 9 \cdot D_e$
7. SolutionSetUpdate $(\gamma, \Gamma', b_0, \alpha)$
8. if the current position allows buffer insertion, then
9. for each $\gamma \in \Gamma'$,
10. for validated buffer type b_i and input slew bin g,
11. set $C(\gamma') = C_{b_i}$
12. set $W(\gamma') = W(\gamma') + W_{b_i}$
13. set $S(\gamma') = 0$
14. set $L_1(\gamma) = \underline{S_g}$
15. set $U_1(\gamma) = \overline{S_g}$
16. SolutionSetUpdate $(\gamma', \Gamma', b_i, \alpha)$
17. // merge Γ_1 and Γ_2 to Γ_{v_t}
18. set $\Gamma_{v_t} = \emptyset$
19. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate γ'
20. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
21. set $W(\gamma') = W(\gamma_1) + W(\gamma_2)$
22. set $S(\gamma') = \max\{S(\gamma_1), S(\gamma_2)\}$
23. set L, U of γ' to be the union of L, U of γ_1, γ_2
24. SolutionSetUpdate $(\gamma', \Gamma_{v_t}, b_0, \alpha)$
25. eliminate infeasible solutions at driver and return γ with the smallest cost

Fig. 7. Slew constrained minimum cost buffering algorithm with non-fixed buffer input slew.

Procedure: SolutionSetUpdate w/ non-fixed input slew
Input: γ': a candidate solution, Γ: a solution set, b: a buffer type, α: a slew constraint **Output:** an updated solution set Γ
1. // check whether γ' violates the slew constraint 2. if $b = b_0$, then 3. return Γ if $S(\gamma') > \alpha$ 4. else 5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$ 6. // domination check and domination elimination 7. for each solution $\gamma \in \Gamma$, do 8. if γ dominates γ' based on (C, W, S, L, U), 9. return Γ 10. if γ' dominates γ based on (C, W, S, L, U), 11. remove γ from Γ 12. insert γ' into Γ and return Γ

Fig. 8. Procedure for updating solution set for slew buffering with non-fixed buffer input slew.

Proof: In essence, the algorithm is only propagating a single candidate up to the source. To insert buffers along a single path, we place a buffer as far (i.e., upstream) as possible from the previously inserted buffer such that the slew constraint is still satisfied. When proceeding to a branching point, a buffer is also placed as upstream as possible while the slew constraint must be satisfied for both branches. It is easy to see that given n buffer positions and sinks, this greedy algorithm returns the optimal solution in $O(n)$ time.

Note that the above greedy algorithm can work in either discrete or continuous case. We now generalize this idea to handle multiple buffer types. As before, we place a buffer as upstream as possible from the previously inserted buffer such that the slew constraint is satisfied. The major difficulty is, of course, every type of buffers can be inserted at a position. Within a single branch, after a new solution is generated (i.e., a buffer is inserted), it is placed into a priority queue, which is decreasingly ordered by the distance from the current buffer position to the root. The first element in the

queue is then extracted as the next solution to be processed. For this solution, all types of buffers are inserted (each of which is placed as upstream as possible) and thus $|B|$ new solutions are generated and placed into the queue. As before, dominated solutions are pruned. For any two solutions γ_1, γ_2 where γ_1 resides at a position no lower than γ_2, γ_1 *dominates* γ_2 if $C(\gamma_1) \leq C(\gamma_2)$, $W(\gamma_1) \leq W(\gamma_2)$ and $S(\gamma_1) \leq S(\gamma_2)$.

Algorithm: Continuous slew buffering w/ fixed input slew.

Input: T, B, α, input slew bins
Output: minimum cost buffer assignment γ satisfying α

```
1.  building solutions at each sink
2.  for each branching point/driver v_t in the order given by
    a postorder traversal of T, let T' be two branches T_1,
    T_2 of v_t and Γ' be the solution set, do
3.      for each γ ∈ Γ', do
4.        for each selected buffer type b_i,
5.          if b_i is necessary up to the branching point,
6.            insert b_i into γ as upstream as possible to obtain γ'
7.          else
8.            propagate γ to v_t by adding wire to obtain γ'
9.        SolutionSetUpdate (γ', Γ', b_i, α)
10.     // merge Γ_1 and Γ_2 to Γ_{v_t}
11.     set Γ_{v_t} = ∅
12.     for each γ_1 ∈ Γ_1 and γ_2 ∈ Γ_2, generate γ'
13.       set C(γ') = C(γ_1) + C(γ_2)
14.       set W(γ') = W(γ_1) + W(γ_2)
15.       set S(γ') = max{S(γ_1), S(γ_2)}
16.       SolutionSetUpdate (γ', Γ_{v_t}, b_0, α)
17. eliminate infeasible solutions at driver and return γ
    with the smallest cost
```

Fig. 9. Continuous slew constrained minimum cost buffering algorithm with fixed buffer input slew.

The above exponential algorithm is found to be inefficient by our experiment. As such, an approximation algorithm through adaptively selecting candidate buffers is proposed. All buffers with area less than a threshold, called *filtered buffers*, are first increasingly sorted according to their slew resistance. For a slew constraint α, the first $\lceil c \cdot (e^{\alpha} - 1) \cdot |B| \rceil$ buffers (note that all $|B|$ buffers will be chosen when the

Procedure: SolutionSetUpdate (continuous, fixed input)

Input: γ': a candidate solution, Γ: a solution set,
b: a buffer type, α: a slew constraint
Output: an updated solution set Γ

1. // check whether γ' violates the slew constraint
2. if $b = b_0$, then
3. return Γ if $S(\gamma') > \alpha$
4. else
5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$
6. // domination check and domination elimination
7. for each solution $\gamma \in \Gamma$, do
8. if γ is at the same position or upstream to γ', and $C(\gamma) \leq C(\gamma')$, $W(\gamma) \leq W(\gamma')$ $S(\gamma) \leq S(\gamma')$,
9. return Γ
10. if γ is at the same position or downstream to γ' and $C(\gamma') \leq C(\gamma)$, $W(\gamma') \leq W(\gamma)$ and $S(\gamma') \leq S(\gamma)$,
11. remove γ from Γ
12. insert γ' into Γ and return Γ

Fig. 10. Procedure of updating solution set for continuous slew buffering with fixed buffer input slew.

value exceeds the number of filtered buffers) are selected to form the library for buffer insertion, where c is a constant and is experimentally determined to be 0.2. One can see that the number of buffer types to be investigated increases exponentially with the slew constraint. The idea behind this selection criterion reads as follows. Roughly speaking, for tight slew constraint, many buffers are needed and there will be many non-dominated solutions. Thus, our computation may only focus on a small number of buffers in order to reduce the size of the solution set. That is, we tradeoff solution quality for runtime. For loose slew constraint, a buffer will be inserted with a large gap from the previously inserted buffer and thus the solution set might not be very large. We can therefore choose more buffers (exponentially more in our case) to obtain high-quality solutions. Varying c, one can achieve different tradeoff between solution quality and runtime. Refer to Figure 9 and Figure 10 for the algorithms for continuous slew buffering with fixed input slew assumption. By combining the techniques in

Section 3, we can easily obtain the algorithms for continuous slew buffering with non-fixed input slew. Refer to Figure 11 and Figure 12 for the algorithms for continuous slew buffering with non-fixed input slew. Finally we present some discussions about bounding the time complexity of the continuous slew buffering algorithm. Existing buffering algorithms often bound their time complexity using the number of candidate buffer positions. This is difficult in our case as candidate buffer positions are not well defined in our continuous buffering problem. Thus, we need to express our time bound using the smallest distance between any of two buffers such that the slew constraint is barely hold. However, bounding the time complexity in this way does not provide much insight in comparing our continuous slew buffering algorithm and other existing algorithms.

5. Buffer Blockage Avoidance

In real circuits, some large area chunks may contain *buffer blockage*, which are macro or IP blocks allowing wire routing but not buffer insertion inside them. As such, a routing tree to be buffered might be re-routed to avoid blockage. For this purpose, we adopt a simultaneous buffer insertion and blockage avoidance approach in [33] which introduces very small additional wire in rerouting while keeps the solution quality. For completeness, we include some details of the approach in [33] here.

It is easy to re-route a path to avoid blockage if it contains no Steiner nodes. Otherwise, we start from the most downstream node inside the blockage, and move each node in turn such that each local move introduces smallest additional wire length. As in [33], we pay attention to the case where no adjustment on topology is needed even if there are Steiner nodes inside the blockage. Suppose that node v moves to v' for blockage avoidance. Downstream solutions need to be propagated to both v and v'. Of course, no buffer can be inserted at v. This propagation process

Continuous slew buffering w/ non-fixed input slew.

Input: T, B, α, input slew bins
Output: minimum cost buffer assignment γ satisfying α

1. building solutions at each sink
2. for each branching point/driver v_t in the order given by a postorder traversal of T, let T' be two branches T_1, T_2 of v_t and Γ' be the solution set, do
3. for each $\gamma \in \Gamma'$, do
4. for each selected and validated buffer type b_i, and for each input slew bin g,
5. if b_i is necessary up to the branching point,
6. insert b_i into γ as upstream as possible to obtain γ'
7. set $C(\gamma') = C_{b_i}$
8. set $W(\gamma') = W(\gamma') + W_{b_i}$
9. set $S(\gamma') = 0$
10. set $L_1(\gamma) = \underline{S_g}$
11. set $U_1(\gamma) = \overline{\overline{S_g}}$
12. SolutionSetUpdate $(\gamma', \Gamma', b_i, \alpha)$
13. else
14. propagate γ to v_t by adding wire to obtain γ'
15. SolutionSetUpdate $(\gamma', \Gamma', b_i, \alpha)$
16. // merge Γ_1 and Γ_2 to Γ_{v_t}
17. set $\Gamma_{v_t} = \emptyset$
18. for each $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$, generate γ'
19. set $C(\gamma') = C(\gamma_1) + C(\gamma_2)$
20. set $W(\gamma') = W(\gamma_1) + W(\gamma_2)$
21. set $S(\gamma') = \max\{S(\gamma_1), S(\gamma_2)\}$
22. set L, U of γ' to be the union of L, U of γ_1, γ_2
23. SolutionSetUpdate $(\gamma', \Gamma_{v_t}, b_0, \alpha)$
24. eliminate infeasible solutions at driver and return γ with the smallest cost

Fig. 11. Continuous slew constrained minimum cost buffering algorithm with non-fixed buffer input slew.

Procedure: SolutionSetUpdate (continuous, non-fixed)

Input: γ': a candidate solution, Γ: a solution set,
b: a buffer type, α: a slew constraint

Output: an updated solution set Γ

1. // check whether γ' violates the slew constraint
2. if $b = b_0$, then
3. return Γ if $S(\gamma') > \alpha$
4. else
5. return Γ if $\sqrt{S(\gamma')^2 + (R_b \cdot C(\gamma') + K_b)^2} > \alpha$
6. // domination check and domination elimination
7. for each solution $\gamma \in \Gamma$, do
8. if γ is at the same position or upstream to γ', and
γ dominates γ' based on (C, W, S, L, U),
9. return Γ
10. if γ is at the same position or downstream to γ' and
γ' dominates γ based on (C, W, S, L, U),
11. remove γ from Γ
12. insert γ' into Γ and return Γ

Fig. 12. Procedure of updating solution set for continuous slew buffering with non–fixed buffer input slew.

continues and eventually, all solutions are merged at the first upstream node (during the bottom-up computation process) outside the blockage. Propagation to both nodes may result in efficient buffer usage. For example, if the original optimal solution for the problem without blockage does not insert any buffer into any blockage, it will still be returned. According to [33], the time complexity for this approach is bounded by $O(ng|B|h^2 + mk)$, where n denotes the number of buffer positions, m denotes the number of sinks, g denotes the maximal candidate solution set size, h denotes the maximal expanded Steiner node set, and k denotes the number of rectangular blockages.

E. Discussion of Related Approaches

1. Minimum Cost Slew Constrained Timing Buffering

We refer to van Ginneken/Lillis' algorithm as *VGL* and the discrete slew buffering algorithm with fixed input slew as *SB*. In order to make a meaningful comparison between them, we first modify VGL to handle a slew constraint, without modifying its delay objective function. The new slew constrained VGL is referred to as *VGL+S*. In this way, we can investigate the difference between simply handling the slew constraint to optimize delay versus handling the slew constraint to optimize cost. For this, the three-tuple (C, W, Q) is augmented to (C, W, Q, S), where Q denotes the required arrival time. Note that domination in timing buffering is defined on C, W, Q but not on S, while S is only responsible for eliminating infeasible solutions. In contrast, domination in slew buffering is defined on C, W, S but not on Q. Therefore, VGL+S algorithm may delete optimal solutions based on timing information while our new algorithm, with domination defined on C, W, S can find the minimum cost solution satisfying slew constraint.

Let us look at a simple example illustrating the difference between timing buffering and slew buffering. Consider merging two solutions sets corresponding to two branches. Suppose that we have two solutions (represented by (C, W, Q, S)) $(3, 30, 70, 20)$, $(5, 50, 80, 10)$ for the left branch, and two solutions $(3, 30, 85, 15)$, $(5, 50, 92, 8)$ for the right branch. Assume that the slew constraint is 50. By timing buffering, we have two non-dominated solutions which are $(6, 60, 70, 20)$ and $(8, 80, 80, 15)$. However, by slew buffering, we have another non-dominated solutions (represented by (C, W, S)) which is $(10, 100, 10)$. Clearly in this case, solutions with a sharper slew rate are not deleted.

The experiments in the next section report the timing-driven buffering solution

as the smallest cost (area) solution at the driver, thereby slack at the driver plays no role. In this way, the impact of the actual change in optimization strategy for area instead of delay is considered.

2. Capacitance-Based Buffering

We also compare slew buffering with another closely related buffering - capacitance-based buffering (CBB) [26, 25]. Roughly speaking, in capacitance-based buffering, downstream capacitance of a buffer cannot exceed the maximum capacitance it can drive, where a single typical buffer is used.

The capacitance-based buffering can be certainly computed in bottom-up fashion. Consider inserting two typical buffers at consecutive nodes v_j (upstream) and v_k (downstream), respectively, and the wirelength in between is the maximum possible value subject to the slew constraint. If the downstream capacitance load at v_j is $C(v_j)$, the capacitance constraint β is set to $\rho C(v_j)$, where ρ is a constant in $(0, 1)$. Note that β is a global constraint in CBB and has a value corresponding to each slew constraint.

F. Experimental Results

1. Experiment Setup

For convenience, all algorithms in comparison are listed below together with their abbreviations.

- SB: discrete slew buffering algorithm with fixed input slew, where input slew is equal to the slew constraint.
- SB+NI: discrete slew buffering with non-fixed input slew.

- C-SB: continuous slew buffering with fixed input slew.
- C-SB+NI: continuous slew buffering with non-fixed input slew.
- SB+B: discrete slew buffering w/ fixed input slew and blockage.
- VGL: van Ginneken/Lillis' min-cost timing buffering algorithm.
- VGL+S: slew constrained VGL.
- VGL+S+PSP: VGL with pre-buffer slack pruning technique [19].
- VGL+S+B: VGL+S with blockage.
- CBB: capacitance-based buffering algorithm.
- CWB: slew constrained buffering with pruning based on (C, W).

All algorithms are implemented in C++ and are tested on a Pentium IV computer with a 3.2GHz CPU and 1G memory. Our test cases are extracted from an industrial ASIC chip, which consist of 1000 nets with more than 50 thousand nodes including sinks, branching nodes and buffer positions. Among them, 757 nets have ≤ 5 sinks and all the remaining nets have ≤ 20 sinks. The sink capacitances range from $2.5fF$ to $200fF$. The wire resistance is $0.56\Omega/\mu m$ and the wire capacitance is $0.48fF/\mu m$. Another set of 100 large-degree nets are used to perform experiments to demonstrate the scalability of the algorithm, where the number of sinks ranges from 105 to 948.

The buffer library consists of 48 buffers, in which 23 are non-inverting buffers and 25 are inverting buffers. Normalized buffer areas range from 5 to 34, slew resistances range from $0.18ns/pF$ to $29.3ns/pF$, and input capacitances range from $2.1fF$ to $76.0fF$.

Table IV. Comparison of discrete slew buffering (SB) and slew constrained timing buffering (VGL+S). #S refers to the average number of non-dominated solutions at driver. Slack is in ns. CPU time is in seconds.

	Discrete Slew Buffering (SB)					Slew Const. Timing Buffering (VGL+S)					Ratio	
Slew (ns)	Area	# Buf	Slack	#S	CPU	Area	# Buf	Slack	#S	CPU	Area	Speed
0.3	44980	7794	8715	71	19.1	46551	9605	8760	271	346.5	3.5%	18.1
0.4	30963	6069	8697	51	15.0	32133	7600	8749	247	351.8	3.8%	23.4
0.5	22960	5108	8511	35	11.7	24235	6858	8613	246	408.1	5.6%	34.9
0.6	18380	4114	8472	27	9.5	19438	5504	8650	254	417.8	5.8%	43.9
0.7	15531	3551	8420	22	8.3	16445	4565	8581	269	463.2	5.8%	55.8
0.8	13340	3216	8387	18	7.5	14218	4300	8542	278	487.1	6.6%	65.0
0.9	11578	2972	8332	14	6.9	12243	3749	8510	292	532.9	5.7%	77.2
1.0	10316	2712	8305	13	6.2	10897	3340	8481	299	548.9	5.6%	88.5

2. Comparison with Timing Buffering

We first compare SB with VGL+S, and results are summarized in Table IV. Here "area saving" refers to the percentage difference in area, "speed up" refers to the percentage difference in CPU time (seconds), and the slew constraint is given in nanoseconds. Note that in SB, the buffer input slew is set to the slew constraint. In VGL+S, range search tree pruning is implemented as in [17]. We make the following observations:

- The number of buffers decreases and the area decreases for both algorithms as the slew constraint loosens. This makes sense since a looser constraint means that buffers can be spaced further apart.

- SB is more efficient in area. For example, with a 1.0 ns slew constraint, the area savings is 5.6% compared to VGL+S.

- The slew buffering algorithm SB is much more efficient. Despite considering all 48 buffers in the library, it runs in just a few seconds on 1000 nets. Furthermore, it runs over 88 times faster than the timing buffering algorithm for slew constraint $\alpha = 1.0$. The main reason for this fact is that there is a signif-

icantly smaller set of non-dominated solutions in slew buffering than in timing buffering. For example, when $\alpha = 1.0$, we have only 13 solutions per net in the slew buffering, while the number is 299 in the slew constrained timing buffering. This is caused by the fact that slew gets to be reset to zero whenever a buffer is inserted, while delay has to be propagated up the entire tree. In practice, the runtime is virtually linear.

- For slew constraint equal to $1ns$, we present a log-log (log (number of buffer positions) v.s. log (CPU time)) plot in Figure 13 where the best linear fit to the data points is also shown. The slope of the linear fit is 1.02. Therefore, the runtime of SB almost linearly depends on the number of buffer positions.

- Comparing slack at driver (c.f. slack degradation ratio in Table IV), one sees that slew buffering achieves significant improvement in runtime with only slight sacrifice in slack. Note that slack here refers to the sum of slacks over all nets.

It is worth mentioning that the range search tree pruning technique, when incorporated into SB, slows down the algorithm as indicated by our experiment. For example, when the slew constraint is 1.0, SB with range search tree returns the solution in 49.8 seconds compared to 6.2 seconds by the one without it. This fact is due to the considerable amount of inherent overhead in maintaining the balanced range search tree data structure.

It is interesting to investigate the following CWB buffering algorithm. In CWB, the pruning condition is based only on (C, W) but not on Q or S. However, S is still maintained throughout solution propagation for checking whether slew constraint is violated. Compared to VGL+S and SB, CWB should certainly run faster since fewer solutions need to be maintained in solution propagation. It is interested to investigate the solution quality degradation by CWB. The results are summarized in Table V.

Comparing Table V and Table IV, one can see that CWB is worse than VGL+S in area. This makes sense by noting the following facts. It is true that both VGL+S and CWB may prune solutions which are actually superior in slew. However, VGL+S maintains much more solutions than CWB and there are some correlations between delay Q and slew S, thus, VGL+S should have larger potential to keep those solutions which are superior in slew. As a result, VGL+S outperforms CWB from about 1% to 5% in buffer area.

Table V. Slew constrained buffering with pruning based on (C, W), CWB. #S: the number of non-dominated solutions at driver. Area Saving is obtained comparing to SB.

	Discrete Slew Buffering based on (C, W)				Ratio
Slew constraint (ns)	Area	# Buf	#S	CPU (s)	Area Sav.
0.3	47993	10265	48	16.0	6.7%
0.4	33848	7793	43	14.2	9.3%
0.5	25210	7058	31	10.9	9.8%
0.6	20019	5591	23	8.5	8.9%
0.7	16730	4582	19	7.9	7.7%
0.8	14283	4398	15	7.1	7.1%
0.9	12358	3850	12	6.5	6.7%
1.0	10985	3379	11	5.9	6.5%

It is known that van Ginneken/Lillis' algorithm is not the most efficient buffering algorithm to handle buffer cost minimization. Several improvements exist. In this paper, we incorporate the pre-buffer slack pruning (PSP) technique proposed in [19] into VGL+S to investigate the performance of SB when compared to one of the state-of-the-art buffering approaches. The results are summarized in Table VI. As one can see from Table VI, SB runs much faster than VGL+S+PSP.

To investigate the scalability of the proposed slew buffering algorithm, experiments on a set of 100 large-degree nets are performed. The number of sinks for these testcases ranges from 105 to 948. As before, we compare SB to VGL+S+PSP and the results are summarized in Table VII. One sees that SB can still run much faster

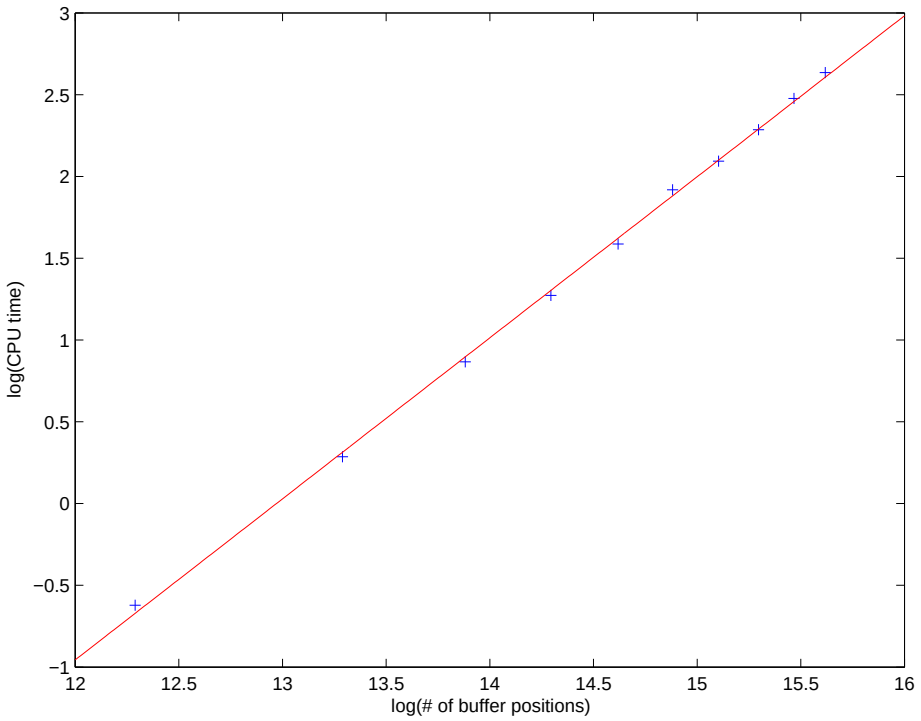


Fig. 13. Illustration of time complexity of SB. +: log (number of buffer positions) v.s. log (CPU time) for slew buffering with slew constraint $1.0ns$. Line: best linear fit.

than VGL+S+PSP while saving areas.

3. Slew Buffering with Non-Fixed Input Slew

Results of SB+NI are summarized in Table VIII. Area saving here refers to comparison to discrete slew buffering with fixed input slew, i.e., SB. We observe the following:

- SB+NI can save up to 21.9% area over SB. In SB+NI, the number of input slew bins to each buffer is 21. For each slew bin, downstream capacitance is also discretized into 21 capacitance bins in the lookup table. With very tight slew constraint, SB+NI saves much more area over SB. It is the case since the actual input slew is significantly smaller than the pre-set upper bound.

Table VI. The comparison of SB and VGL+S+PSP (VGL+S incorporated with pre-buffer slack pruning [19]). Speed up refers to the runtime difference between SB and VGL+S+PSP.

Slew constraint	CPU (s) of VGL+S+PSP	Speed up
0.3	269.3	14.1
0.4	303.0	20.2
0.5	338.1	28.9
0.6	365.7	38.5
0.7	374.3	45.1
0.8	433.5	57.8
0.9	478.2	69.3
1.0	487.9	78.7

Table VII. Comparison of discrete slew buffering (SB) and slew constrained timing buffering (VGL+SB+PSP) on 100 large-degree nets. Slack is in ns.

	Discrete Slew Buffering (SB)					Timing Buffering w/ PSP					Ratio	
Slew (ns)	Area	# Buf	Slack	#S	CPU	Area	# Buf	Slack	#S	CPU	Area	Speed
0.3	73873	12082	2102	532	373.2	75532	14293	2130	1692	2127.3	2.2%	5.7
0.4	61082	11839	2091	438	285.3	62323	13992	2151	1750	2230.0	2.0%	7.8
0.5	43992	9870	2130	359	203.9	45340	12840	2172	1779	2292.2	3.1%	11.2
0.6	38918	8378	2117	310	149.0	40670	12058	2185	1812	2495.1	4.5%	16.7
0.7	32538	7189	2055	257	101.7	34132	11582	2138	1830	2553.5	4.9%	25.1
0.8	25127	6032	2050	215	85.5	26615	8172	2125	1855	2628.2	5.9%	30.7
0.9	23295	5578	2043	190	77.6	24879	7220	2118	1873	2858.7	6.8%	36.8
1.0	20152	4902	2032	157	73.0	21582	6175	2111	1891	3029.2	7.1%	41.5

- SB+NI becomes slower with tighter slew constraint since the size of the solution set becomes much larger as more buffers are inserted.

- For speedup, we apply the adaptive buffer selection technique as described in Section D for continuous slew buffering to SB+NI. From Table VIII, one can see that SB+NI is significantly accelerated while the solution quality is moderately degraded.

4. Continuous Slew Buffering

Results of C-SB and C-SB+NI are summarized in Table IX. Area saving here refers to comparison to SB. We observe the following:

Table VIII. Results of slew buffering with non-fixed input slew. Area saving is obtained by comparing to SB. Degrad. refers to the slack degradation obtained by comparing to VGL+S.

	Slew Buffering w/ Non-Fixed Input Slew (SB+NI)					SB+NI w/ Adaptive Buffer Selection				
Slew	Area	# Buf	CPU	Degrad.	Area Saving	Area	# Buf	CPU	Degrad.	Area Saving
0.3	35148	7114	992.1	17.4%	21.9%	37717	7075	19.2	11.7%	16.2%
0.4	25018	5666	931.7	12.3%	19.2%	31661	5000	27.0	6.6%	-2.3%
0.5	19797	4326	762.8	8.0%	13.8%	23841	4175	39.8	1.6%	-3.8%
0.6	16528	3772	569.3	4.6%	10.1%	18921	5406	54.9	1.3%	-2.9%
0.7	13995	3463	473.6	5.1%	9.9%	14941	4958	69.3	2.1%	3.8%
0.8	12129	3145	397.4	4.6%	9.1%	12178	3214	78.2	3.5%	8.7%
0.9	10667	2854	365.2	4.6%	7.9%	10736	2952	81.3	4.3%	7.3%
1.0	9629	2488	337.3	3.9%	6.7%	9663	2525	85.0	2.4%	6.3%

- In slew buffering, tighter constraint causes excessive buffer insertion. If the candidate buffer positions are not pre-set carefully in discrete slew buffering, we may often have to insert buffers in an inefficient way. Continuous slew buffering (C-SB) significantly alleviates this problem and results in up to 15% improvement in buffer area.

- C-SB runs very fast due to our adaptive procedure for buffer selection (see Section D). If C-SB is carried out without buffer selection procedure, the algorithm becomes very slow. For example, we obtain a solution with buffer area 9703 in 1722.8 seconds for $\alpha = 1.0$. Compared to C-SB with buffer selection, it is only 0.5% better in buffer area, however, it is about 75× slower.

- We also implement the continuous slew buffering without fixed input slew assumption (C-SB+NI). As is evident from Table IX, one can further save several percentage area over C-SB while the runtime is still acceptable.

Table IX. Results of continuous slew buffering. Area saving is obtained by comparing to SB. Degrad. refers to the slack degradation obtained by comparing to VGL+S.

	Continuous Slew Buffering (C-SB)					C-SB w/ Non-Fixed Input Slew (C-SB+NI)				
Slew	Area	# Buf	CPU	Degrad.	Area Saving	Area	# Buf	CPU	Degrad.	Area Saving
0.3	37840	6383	2.5	9.8%	15.8%	37627	6149	19.7	15.4%	16.4%
0.4	27905	4717	2.7	5.7%	9.9%	24927	4980	285.7	8.8%	19.5%
0.5	21043	4202	9.3	5.9%	8.3%	19281	3851	220.6	7.4%	16.0%
0.6	16880	4735	40.9	6.5%	8.9%	15831	4622	710.5	5.7%	13.9%
0.7	14525	3420	33.1	5.2%	6.9%	13017	3190	700.3	5.2%	16.2%
0.8	12472	3198	29.2	5.9%	6.5%	10813	2886	610.7	6.6%	18.9%
0.9	10872	2883	25.8	5.1%	6.1%	9582	2461	532.8	6.2%	17.2%
1.0	9754	2630	22.9	5.7%	5.4%	8768	2277	442.1	5.5%	15.0%

5. Handling Blockage

The experimental results on discrete slew buffering with blockage (SB+B) and the slew constrained timing buffering with the same blockage (VGL+S+B) are included in this paper. Note that SB+B holds the fixed input assumption. We randomly place 20 rectangular blockages with total area summed to 30% that of the smallest bounding box of each net. Results are shown in Table X. Since blockages are introduced, solution quality of both slew buffering and timing buffering becomes worse than before, i.e., area saving is negative. However, slew buffering still outperforms timing buffering in terms of both runtime and buffer area.

It is interesting to see that timing buffering tends to have more computation overhead than slew buffering when buffer blockages are handled. We would like to interpret this phenomenon as follows. VGL+S is very sensitive to buffer positions in terms of runtime since a new buffer position may lead to many new non-dominated solutions (see Section b). However, in slew buffering, a new buffer position can only lead to $|B|$ new solutions as discussed in Section b. In fact, the slew buffering algorithm runs almost linear in the number of buffer positions as indicated by Figure 13. Thus, slew buffering is less sensitive to the buffer blockage insertion in terms of runtime.

Table X. Handling blockage. Each net has 30% blockage area. Area saving is obtained by comparing to SB.

	Slew Buffering w/ Blockage				Timing Buffering w/ Blockage			
Slew constraint (ns)	Area	#Buf	CPU (s)	Area Saving	Area	#Buf	CPU (s)	Area Saving
0.3	51347	8502	22.3	-12.4%	52121	10792	423.7	-13.7%
0.4	35467	6792	19.3	-12.7%	36214	8302	447.7	-14.5%
0.5	26180	5893	16.2	-12.3%	26635	7693	470.3	-13.8%
0.6	20863	4721	13.2	-11.9%	21201	5887	542.3	-13.3%
0.7	17831	4082	10.0	-12.9%	18380	4952	610.1	-15.5%
0.8	15073	3529	9.2	-11.5%	15731	4598	662.3	-15.2%
0.9	13202	3290	8.7	-12.3%	13767	4110	710.5	-15.9%
1.0	11845	3021	7.9	-12.9%	12237	3775	759.0	-15.7%

6. Comparison with Capacitance-Based Buffering

Finally, we compare SB with CBB [26, 25]. As in practice, a typical buffer is selected for running CBB. As such, we calculate for each buffer b_i the longest wire length l_i it can drive such that the slew constraint is satisfied. The typical buffer is the one with maximum $l_i/A(b_i)$ value, where $A(b_i)$ is the buffer area of b_i. The data in Table XI (c.f. Table IV) demonstrate that SB significantly outperforms CBB in our context: the total area of solutions by CBB is usually more than double that of SB. The area of capacitance based buffering is much worse because only a single buffer is used, capacitance based buffering ignores resistive effect, and multi-pin nets are not well handled in CBB. Note that CBB runs in less time since only a single buffer is used in computation.

Table XI. Capacitance-based buffering (CBB). Only a single typical buffer is used. Area saving is obtained by comparing to SB.

Slew (ns)	Area	# Buf	CPU (s)	Area Saving
0.3	86572	12357	1.1	-48.0%
0.4	62101	8864	1.3	-50.1%
0.5	54324	7754	1.5	-57.7%
0.6	49825	7112	1.5	-63.1%
0.7	42526	6070	1.4	-63.5%
0.8	36956	5275	1.3	-63.9%
0.9	31611	4512	1.0	-63.4%
1.0	26447	3775	0.9	-61.0%

G. Conclusion

This work proposes a new buffering formulation motivated by the need to efficiently buffer huge numbers of nets under slew constraints. We show that one can optimize for area and satisfy a slew constraint efficiently, despite the problem being NP-hard.

The slew buffering problem is intensively studied in this work. Three new algorithms are proposed, namely, a slew buffering algorithm with the assumption of fixed input slew, a more sophisticated algorithm without this assumption, and a very efficient continuous slew buffering algorithm. Experimental results demonstrate that new algorithms run one to two orders of magnitude faster than the widely-used timing buffering algorithm and meanwhile they can obtain significant amount of area saving.

CHAPTER III

GATE SIZING FOR CELL LIBRARY-BASED DESIGNS

With increasing time-to-market pressure and shortening semiconductor product cycles, more and more chips are being designed with library-based methodologies. In spite of this shift, the problem of discrete gate sizing has received significantly less attention than its continuous counterpart. On the other hand, cell sizes of many realistic libraries are sparse, for example, geometrically spaced, which makes the nearest rounding approach inapplicable as large timing violations may be introduced. Therefore, it is highly desirable to design an effective algorithm to handle this discrete gate sizing problem.

Such an algorithm is proposed in this paper. The algorithm is a continuous solution guided dynamic programming approach. A set of novel techniques, such as Locality Sensitive Hashing based solution pruning, are also proposed to accelerate the algorithm. Our experimental results demonstrate that (1) nearest rounding approach often leads to large timing violations and (2) compared to the well-known Coudert's approach, the new algorithm saves up to 21% in area cost while still satisfying the timing constraint.

A. Introduction

Increasing design complexities along with time-to-market pressures and shortening product cycles have mandated a shift in VLSI design from custom crafting to cell library-based design methodologies. This shift raises an increasing need of salient gate sizing techniques which are powerful in performing delay-area trade-off optimizations. A handful set of gate sizing techniques exist, however, most of them handle the continuous gate sizing problem which is based on the assumption that gate sizes can

be any values within certain range (see, e.g., [34, 35, 36]). When gate implementations are restricted to discrete sizes, as in reality, the problem becomes much more difficult and very few approaches (see, e.g., [37, 6]) are known.

On the other hand, a large number of realistic cell libraries are "sparse". For example, when the cell sizes are geometrically spaced instead of uniformly spaced, significant sparseness is introduced. Refer to [7] for some realistic sparse libraries. Geometrically spaced gate sizes are desired because uniformly spaced gate sizes would result in a large number of gate sizes and managing this large volume of data is difficult [7]. Furthermore, it is proven in [7] that under certain conditions, the set of optimal gate sizes must satisfy the geometric progression.

In this work, we propose a novel gate sizing technique which handles discrete gate sizes. As many efficient solutions exist for the continuous gate sizing problem, one might think of obtaining a discrete solution through rounding a continuous solution. This is very fast but often results in large timing violations for a sparse cell library. In contrast, the method proposed by Coudert [6], which is based on the multi-dimensional descent optimization, handles the discrete sizes. However, it has some trial-and-error flavor and thus has room for further improvement. A dynamic programming approach can search solutions more systematically and therefore has the potential to generate high quality solutions. However, it may suffer from substantial amount of computation overhead, which imposes a great challenge to our problem.

The key idea of the new algorithm is to integrate the solution quality of dynamic programming with the short runtime of obtaining solution to the continuous version of the problem. That is, we narrow down the searching space of dynamic programming under the guidance from a best continuous solution. Thus, instead of checking every implementation, our algorithm only investigates a number of discrete implementations around the best continuous solution. This enables us to find so-

lutions with quality close to the best continuous case and at the same time obtain huge speedup in computation. We also develop new techniques to prune inferior solutions and maintain/increase the diversity of intermediate solutions. Focusing on a small number of diversified and representative solutions during the candidate solution propagation can improve the efficiency of solution search. To this end, an advanced scheme called *Locality Sensitive Hashing* (LSH) technique [38] is explored to maintain solution diversity via selecting well-spaced solutions in high dimensions.

In summary, the main contributions of this paper are (1) a continuous solution guided dynamic programming approach for discrete gate sizing, and (2) a Locality Sensitive Hashing based solution pruning technique that allows obtaining high quality solutions by maintaining solution diversity.

Our experimental results demonstrate that (1) nearest rounding approach often leads to large timing violations for sparse cell libraries and (2) compared to the well-known Coudert's approach in [6], the new algorithm saves up to 21% area cost for basic library while still satisfying the timing constraint. For a sparser library the new algorithm provides benefits of up to 24% in area cost, thus showing that sparser the library better are the benefits of the new algorithm.

The rest of the chapter is organized as follows. Section B presents the problem formulation. Section C describes the algorithm for discretizing the continuous solution. Sections E presents the experimental results. A summary of work is given in Section G.

B. Problem Formulation

Given a combinational circuit with n gate nodes, n_i primary input nodes and n_o primary output nodes, a gate library L consisting of $|L|$ gate types where each gate type,

characterized by the functionality and the number of gate inputs, may have various gate sizes, the *discrete gate sizing* problem asks to compute a sizing solution with the *minimal total gate area cost* such that the maximum delay between any primary input node and any primary output node is bounded above by a delay constraint α. The problem can be formally defined as

$$\begin{array}{ll} Minimize & \sum_{i=1}^{n} a_i W_i \\ s.t. & Delay \leq \alpha, \\ & W_i \in L, \end{array} \tag{3.1}$$

where W_i denotes the size of gate i and a_i denotes its weighting factor. Weighting factors can be set to unity for gate size minimization, and to weighted summation of signal probabilities and activity factors for explicit power optimization.

C. Optimization Methodology

Before investigating the discrete gate sizing problem, it is helpful to go over the closely-related continuous gate sizing problem. In this problem, gate sizes are allowed to be any real value between certain lower and upper bounds, and the resulting problem can be efficiently solved, for example, by using Lagrangian Relaxation technique [35]. Moreover, an optimal solution can be obtained if the underlying delay model is a posynomial function. For example, the solution is proven to be optimal when the Elmore delay model is used [35].

1. Error Due to Nearest Rounding

It might be expected that a good discrete solution can be obtained by rounding the gate sizes of continuous solution to the nearest discrete gate sizes. However, this is not the case for the sparse cell library as the choices of gate implementations are

very restrictive (see also [6] for this observation). We have the following theoretical analysis on error bound of nearest rounding.

Suppose that gate sizes are geometrically spaced with factor t for each gate type, i.e., available gate sizes are as $1, t, t^2, t^3, \ldots$ for each gate type. Given any continuous solution, the timing after nearest rounding is bounded above by $t\times$ the timing of the continuous solution. For example, if the timing of the continuous solution is $1ns$ and $t = 2$, then after nearest rounding, the new timing is at most $2ns$.

In our proof, as in [35], for a gate with size x, gate capacitance is $C(g) = c_u x + c_f$ and gate resistance is $R(g) = r_u/x$, where c_u, c_f, r_u are unit size gate capacitance, gate perimeter capacitance, and unit size gate resistance. We are to bound the maximum delay increase for a single gate due to nearest rounding. For any gate g, nearest rounding can make the resistance at most $(1+t)/2\times$ the resistance of the continuous solution. This happens when the continuous gate size is $(1+t)/2 \cdot t^\alpha - \epsilon$ where α is any non-negative integer and ϵ is a very small positive value. This gate will be rounded down to the size of t^α. Denote by R^c the resistance of the continuous gate, and thus $R^c(g) = r_u/((1+t)/2 \cdot t^\alpha - \epsilon)$. After nearest rounding, the new resistance is $R(g) = r_u/(t^\alpha) \approx (1+t)/2 \cdot R^c(g)$. Suppose that each fanout gate g_i of g is of size $(1+t)/2 \cdot t^\alpha + \epsilon$ and thus they will be rounded up to size $t^{\alpha+1}$. Denote by C^c the capacitance of a continuous fanout gate, and $C^c(g_i) = c_u((1+t)/2 \cdot t^\alpha + \epsilon) + c_f$. After nearest rounding, the new capacitance is $C(g_i) = c_u(t^{\alpha+1}) + c_f \leq 2t/(1+t) \cdot C^c(g_i)$. Denote by $C^c_{downstream}$ the total downstream capacitance of gate g in the continuous solution, then the new total downstream capacitance $C_{downstream}(g) \leq 2t/(1+t) C^c_{downstream}(g)$. Clearly, the delay for the gate g can increase at most to $R(g) \cdot C_{downstream}(g) \leq tR^c(g) \cdot C^c_{downstream}(g)$. Since the gate delay of any gate can be at most increased to $t\times$ the gate delay in the continuous solution, the delay increase for the whole circuit is also at most $t\times$. However, we understand that there is a bit of pessimism built in

this bound, since the bound requires each gate's delay become $t\times$ the delay in the continuous scenario. This cannot happen because if all outputs of a gate are upsized for the gate delay to become $t\times$, then the delay of the next gate on the critical path does not become $t\times$ since it will also be upsized. As a result, delay of the whole path does not become exactly $t\times$ the delay in continuous scenario, but somewhat smaller than this number.

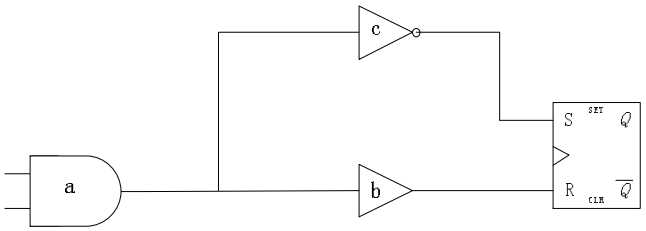


Fig. 14. An example for illustrating rounding error bound due to nearest rounding.

On the other hand, in theory, $t\times$ timing can be obtained by nearest rounding. Let us look at a simple example for this. Refer to Figure 14 where the combinational circuit contains three gates a, b, c. Assume that $c_u(a) = 1, r_u(a) = 1, c_u(b) = 1, r_u(b) = 1$, $c_u(c) = 100, r_u(c) = 0.01$, $c_f(a) = 0, c_f(b) = 0, c_f(c) = 0$, and input capacitance of the flip-flop is 1. Suppose that gate sizes are geometrically distributed with factor of 2. In a continuous solution, suppose that a, b, c are of sizes $1.5 - \epsilon, 1.5 - \epsilon, 1.5 + \epsilon$, respectively. Clearly, $a - b - flipflop$ is the critical path which has delay of about 100. a, b, c will be rounded to $1, 1, 2$, respectively, due to nearest rounding, and $a-b-flipflop$ is still the critical path which has delay of about 200. Thus, the delay is increased to about $2\times$ due to nearest rounding. We reach the following theorem.

Theorem 1: For a cell library where gate sizes are geometrically spaced with factor t, the timing due to nearest rounding can be at most $t\times$ the timing of the continuous solution.

It is interesting to note that nearest rounding may get worse results with denser

cell library. Suppose that in Figure 14, two cell libraries, one sparser library and one denser library, are available. The gate sizes in the sparser library are geometrically distributed with factor of $t = 2$ (i.e., $1, 2, 4, 8, \ldots$). In the denser cell library, there is a gate size $(1 + t)/2 \cdot t^{\alpha}$ for every α. Thus, the gate sizes are distributed as $1, 1.5, 2, 3, 4, 6, 8, \ldots$. Suppose that in a continuous solution, a, b, c are of sizes $1.5 + \epsilon, 1.5 - \epsilon, 1.5 - \epsilon$, respectively. With the sparser cell library, a, b, c will be rounded to $2, 1, 1$, respectively. Thus, the circuit delay is about 50. With the denser cell library, a, b, c will be rounded to $1.5, 1.5, 1.5$, respectively. The circuit delay is then about 100. Clearly, nearest rounding obtains better timing and area with sparser library compared to denser library. Note that this observation does not contradict with Theorem 1. A circuit could have larger timing violation with denser cell library due to nearest rounding as long as its timing violation is smaller than the bound in Theorem 1.

In addition to the theoretical analysis, our study shows that in a sparse cell library, the nearest rounding strategy can make the ISCAS'85 benchmark circuits to have timing violations of hundreds of picoseconds in $90nm$ technology.

2. Proposed Methodology

Nearest rounding may introduce significant amount of timing violations for sparse cell library. On the other hand, the dynamic programming approach can obtain the optimal solution for the discrete gate sizing problem, however, it is computationally prohibitive as it needs to investigate every gate size at each gate node.

Our idea is to integrate the optimality of the dynamic programming framework with the high efficiency of obtaining the solution to the continuous version of the problem. To this end, we propose a *continuous solution guided dynamic programming algorithm* to solve the discrete gate sizing problem. That is, the searching space of

the dynamic programming is significantly narrowed down under the guidance from a good continuous solution while solution quality is only slightly degraded in spite of huge speedup. At each gate node, instead of every discrete gate size, only those close to the continuous solution will be investigated. This difference between the dynamic programming approach (without any speedup technique) and the new scheme enables us to find solutions with quality close to the best continuous case and at the same time obtain tremendous speedup in computation.

There are many continuous gate sizing techniques which use various delay models such as Elmore delay model in [35] and convex delay model in [39]. For simplicity, we adopt Elmore delay model in this paper. However, our method is independent of delay model and any delay model is applicable. For example, the convex delay model [39] can be employed to compute a better continuous solution guider and a better overall result. Rest of the paper concentrates on our novel algorithm used to discretize the continuous solution.

D. Discretization Algorithm

Before explaining the algorithm, we will present our circuit model and elaborate on some key terms necessary to describe the algorithm. In our algorithm, a circuit is represented as a directed acyclic graph where each node corresponds to either a logic gate or a primary input or output of the circuit, and each edge corresponds to a pin-to-pin connection between two gates.

In this paper, a *complete solution* refers to the determination of all gate sizes in the circuit. A *partial solution* is a solution where not all gate sizes have been determined. A partial solution becomes a complete solution when all gates are processed. For convenience, when there is no confusion, we also call a partial solution a solution.

We denote a discrete solution by γ and the underlying continuous solution by γ^c. Outline of the algorithm is described in Figure 15.

Algorithm: Discretize_GateSize.
Input: Continuous solution γ^c **Output:** Discrete solution
1. for eāch gate k during the breadth-first traversal of the circuit graph 2. try several discrete gate sizes around its continuous gate size based on k's criticality 3. generate partial solutions and perform node pruning 4. if the size of the solution set is greater than a threshold 5. perform solution set pruning 6. select the best solution at the primary output

Fig. 15. Pseudocode for discretization algorithm

The algorithm begins with the primary input nodes, proceeds through a breadth-first traversal of the circuit graph and processes each gate in turn. As our discretization approach is guided by the continuous solution, at each gate node, only discrete gate sizes close to the continuous solution are investigated. To this end, each gate is measured by its criticality and more gate sizes will be investigated for those more critical gates. The details of this step are presented in Section 1.

During breadth-first traversal, once a gate node is *processed* it is included in a partial solution. Many partial solutions may be formed. It is necessary to perform solution pruning technique to reduce the size of the solution set and thus save the runtime. There are two types of pruning. The first type of pruning is called node pruning which is performed during the time a node is processed. The second type of pruning is called solution set pruning which is performed after a node is processed and when the number of partial solutions is greater than a threshold. The threshold is experimentally determined to achieve balance between solution quality and runtime for each circuit. This solution set pruning technique uses locality sensitive hashing technique and cutline pruning technique to effectively and efficiently reduce the size of the solution set. These two types of pruning are explained in Sections 2 and 3

respectively.

1. Explore Gate Sizes Close to the Continuous Solution

During the breath-first traversal, the gate sizes investigated at each gate are determined by the criticality of the gate. In our algorithm, criticality of a gate is measured by its slack, namely, a gate is more critical if its slack is smaller. Since one does not know the slack of each node before completing the sizing procedure, the slack is estimated using our guider, i.e., the continuous solution. We first identify the worst slack, denoted by WS, of the circuit in the continuous solution, and then the criticality of each gate g, denoted by $Criticality(g)$, is measured by $Slack(g)/WS$. Note that when the worst slack is very close to zero, a small positive constant ϵ is added to the slack of all nodes for the robustness of computation. Thus, $Criticality(g) = (Slack(g) + \epsilon)/(WS + \epsilon)$.

Our approach is a criticality-based approach, meaning that we spend more discretization efforts on more critical gates. In this paper, we implement this idea by classifying gates into three groups and spend different amount of discretization efforts for each group. Note that our approach is not restricted to three groups and other classification methods can be used. In this paper, we set up two thresholds t_1, t_2 where $1 < t_1 < t_2$ such that when $Criticality(g) < t_1$, T_1 sizes around g's continuous gate size are explored, when $t_1 \leq Criticality(g) < t_2$, T_2 sizes around g's continuous gate size are explored, and when $Criticality(g) \geq t_2$, only the gate size closest to g's continuous gate size is explored. Thus, gate criticality decides how many sizes close to the continuous solution are explored. By this, the search space is greatly reduced in our discrete gate sizing algorithm compared to the standard dynamic programming approach.

2. Solution Pruning

During solution propagation, even though the search space is judiciously restricted by the use of the continuous guider, at each gate, the algorithm still needs to check and keep several solutions close to the continuous solution. Propagating all solutions is impractical and unnecessary. Many of them are *inferior* to others and should be pruned to save computation cost. There are two types of pruning/inferiority in our case. The first type of pruning is called node pruning which is performed during the time a node is processed. The second type of pruning is called solution set pruning which is performed after a node is processed and when the size of the solution set is greater than a threshold.

We now introduce the first type of pruning. Each solution is characterized by cumulative delay and cumulative gate area. That is, each solution γ is characterized by a (D, W) pair, where $D(\gamma)$ refers to the maximum delay from any primary input node to any processed node in γ and $W(\gamma)$ refers to the cumulative gate area for all processed gates in γ. After processing a node, (D, W) will be accordingly updated for each new solution. Note that computing D may require the knowledge of downstream gates (i.e., input capacitance of downstream gates) which are not yet processed. The continuous gate sizes at those downstream gates are then used as an approximation. This makes sense as our whole approach is guided by the continuous solution.

For two solutions γ_1, γ_2 generated at *the same* input gate g_i, γ_2 is inferior to γ_1 if and only if $D(\gamma_1) \leq D(\gamma_2)$ and $W(\gamma_1) \leq W(\gamma_2)$. The inferior solution will be pruned and thus only the solution with either smaller maximum delay or smaller total area survives. This type of pruning is called *node pruning*.

After carrying out the computation for a while, the size of the solution set Γ becomes very large. To maintain efficiency, Γ has to be shrunken before successive

computations. For this purpose, when $|\Gamma|$ is greater than a threshold (after processing a gate node), the second type of pruning, called *solution set pruning*, is performed. In this pruning technique, we also consider to maintain the diversity of the solutions. This is desired since "widely spread" solutions may enable us to search in a large space and eventually lead to better solutions. Focusing on a small number of diversified and representative solutions can improve the efficiency of solution search: we do not want to waste time on checking many similar solutions.

To diversify solutions, we group similar solutions and then select the representative one from each group for further propagation. The solutions which have not been selected are pruned to save runtime. Suppose that we have computed clusters among solutions and inside each cluster, solutions are similar to each other. In each cluster, the representative solution is the one closest to our continuous guider. The representative solutions will be selected for further propagation while other solutions are pruned.

The proximity of a partial discrete solution to the continuous solution is defined by both delay and area information. We first define the concept of a cut line, as shown in Figure 16, as a subset of edges in the circuit such that it partitions the circuit graph into two disjoint subgraphs. Given the current solution set (note that all solutions have the same set of processed gates), the cut line is formed such that for each edge cut by the cut line, it links two gate nodes where the upstream (i.e., fanin) gate node has been processed but the downstream (i.e., fanout) gate nodes has not been processed. Denote all those fanin gate nodes by $\{g_i\} = \{g_1, g_2, \ldots\}$. Each solution γ is assigned a proximity value $f(\gamma)$ which is defined as

$$f(\gamma) = \sum_{g_i} |D(\gamma(g_i)) - D(\gamma^c(g_i))| \cdot \sum_{g_i} |W(\gamma(g_i)) - W(\gamma^c(g_i))|, \tag{3.2}$$

where $|D(\gamma(g_i)) - D(\gamma^c(g_i))|$ measures the delay difference between γ and the contin-

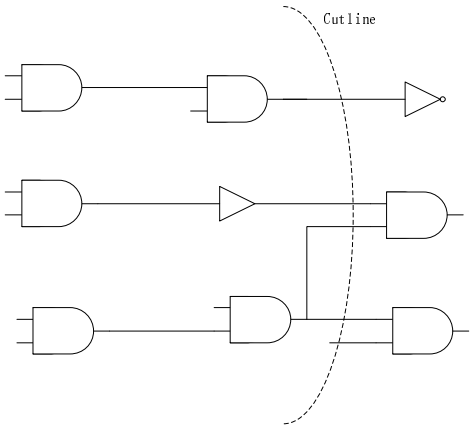


Fig. 16. A cutline.

uous solution and $|W(\gamma(g_i)) - W(\gamma^c(g_i))|$ measures the area difference. Clearly, the f value of a solution measures its proximity to the continuous solution. Suppose that there are many nodes along the cutline. It is difficult and inefficient to compare all solutions by comparing each node along the cutine. The proximity defined as above assigns a single value for each solution which makes the comparison much easier and efficient. Clearly, a solution with small proximity value is preferred. In another word, γ_2 is inferior to γ_1 if and only if $f(\gamma_1) \leq f(\gamma_2)$. The representative solution from each cluster is the one which smallest proximity value.

We are to describe the solution grouping technique. Grouping solutions can be realized through mapping each solution to a high-dimensional vector followed by clustering geometrically close vectors. The mapping is computed as follows.

Assume that each gate node in the circuit is indexed and each gate implementation in the gate library is also indexed, then each gate index corresponds to a distinct dimension and the coordinate along that dimension is equal to the index of the assigned gate implementation. In this way, a solution is mapped to a d dimensional vector if d nodes have been processed. We are now to compute clusters among the resulting vectors. For a large circuit, one has to cluster vectors in a very high dimen-

sion. Although the high-dimensional clustering problem has been intensively studied in decades, it remains as one of the hardest problems in the database research. In this paper, based on a highly effective and efficient nearest neighbor query technique called Locality Sensitive Hashing (LSH) [38], a new technique for solution clustering and representative solution selection is introduced.

3. Solution Clustering by LSH

In cluster computation and nearest neighbor query, "curse of dimensionality", which roughly says that the computational complexity of the above two operations increases exponentially with dimension, remains as a notorious problem in database research for a long time. To effectively attack this problem, a new approach, called *Locality Sensitive Hashing* (LSH), is introduced in [38]. With provable bound on approximations, LSH can efficiently approximate similarity search by hashing technique. The basic idea is to hash the vectors such that the geometrically close (resp. far apart) vectors are hashed to the same (resp. different) bins with large probability. LSH enables us to answer a nearest neighbor query in $O(dm^{1/(1+\epsilon)})$ time over an m-point d-dimensional database for any $\epsilon > 0$. In addition, an approximate nearest neighbor query can be answered in sublinear time *excluding the preprocessing time* [38], where given a point set P, the problem asks to return a point $p \in P$ such that the distance of p to the query point q is at most $1 + \epsilon$ times the distance from the nearest point in P to q. It is shown in [38] that LSH is much more efficient compared to many other methods. Successful applications of LSH include [40] on bioinformatics. LSH can be easily used for clustering since each bin in hash table can be treated as a cluster (note that geometrically close vectors are hashed to the same bin with large probability).

Suppose that d nodes have been processed in each solution in the solution set Γ. Each solution γ is first mapped to a d-dimensional point p, where a dimension

corresponds to a node. Suppose that there are m solutions in the solution set. After mapping, there are m d-dimensional points, which form the point set P. We then embed these points into the Hamming space H with dimension $d' = Md$, where M is the number of available sizes for any cell type in the library. This embedding allows us to perform random sampling on the embedded bit string. For embedding, taking each point $p \in P$, we transform it into a binary vector $v(p) =< \Upsilon_M(x_1), \ldots, \Upsilon_M(x_d) >$ where $\Upsilon_M(x)$ denotes the unary representation of x (i.e., x ones followed by $M - x$ zeroes).

v (p1)	10000:11000:11111
v (p2)	10000:10000:11111
v (p3)	11000:10000:11100

(a) original three bit strings from $v(p)$.

v' (p1)	0111
v' (p2)	0011
v' (p3)	1000

(b) bit strings $v'(p)$ after removing redundancy.

h (p1)	01
h (p2)	01
h (p3)	10

(c) bit strings $h(p)$ after locality sensitive hashing.

Fig. 17. Illustration of concepts in LSH.

It is helpful to illustrate the above concept by a simple example. Suppose that in a solution set Γ, there are three solutions $\gamma_1, \gamma_2, \gamma_3$ and three nodes have been processed. Supposed in solution γ_1, the processed nodes are assigned with sizes of $1, 2, 5$, respectively, which are the indices of the sizes for each processed gates. The solution is then mapped to a point $p_1 = (1, 2, 5)$. Further suppose that $M = 5$, then $|v(p_1)|$ has length of $d' = 15$ and $v(p_1) = 10000 : 11000 : 11111$ since e.g., $2 = 11000$ in unary representation (i.e., 2 ones followed by 3 zeros). Similarly, suppose that in

γ_2, the assigned gate sizes are $1, 1, 5$, and in γ_3, the assigned gate sizes are $2, 1, 3$. γ_2 will be mapped to a point $p_2 = (1, 1, 5)$ and $v(p_2) = 10000 : 10000 : 11111$. γ_3 will be mapped to a point $p_3 = (2, 1, 3)$ and $v(p_3) = 11000 : 10000 : 11100$. Refer to Figure 17(a) for the result.

For convenience, $v(p)$ for a point p is treated as a bit string. It is clear that $v(p)$ can be very long for the large circuit. Given a set of solutions $\{\gamma_1, \gamma_2, \ldots\}$ and their corresponding set of $\{v(p_1), v(p_2), \ldots\}$, many bits may be the same. In a solution set, each solution is characterized only by its difference from other solutions. Thus, we can remove the redundancy before clustering them for high efficiency. For this purpose, $\{v(p_1), v(p_2), \ldots\}$ are reduced to $\{v'(p_1), v'(p_2), \ldots\}$ by keeping only the difference among solutions. Denote the length of $v'(p)$ by $\mathtt{d}$.

In Figure 17(a), one sees that in $\{v(p_1), v(p_2), v(p_3)\}$, only four bits are different and all other bits are the same. Thus, we can shrink the length of the bit strings to four. Refer to Figure 17(b) for the result.

LSH then performs a further dimension reduction mapping to the bit strings through random sampling for clustering. For dimension reduction mapping, we randomly choose k elements from $\{1, 2, \ldots, \mathtt{d}\}$, where each element has equal probability to be chosen, and form an index subset $I = \{i_1, i_2, \ldots, i_k\}$. We then map each point p into $h(p) =< v'(p)[i_1], v'(p)[i_2], \ldots, v'(p)[i_k] >$.

For Figure 17(b), if we choose $k = 2$ elements from $\{1, 2, 3, 4\}$ as $1, 4$, then $I = \{1, 4\}$ and $h(p_1) = 01$ since $v'(p_1)[1] = 0$ and $v'(p_1)[4] = 1$. Similarly, $h(p_2) = 01$ and $h(p_3) = 10$. Refer to Figure 17(c) for the result.

Function $h(\cdot)$ is called the *locality-sensitive hash function* [38, 40]. It is shown in [38] that the probability for two embedded points to have the same hash value (i.e., hashed into the same bucket) is "proportional" to their similarity. Based on this intuitive fact, we are able to build hash table to support efficient similarity search

and clustering among a set of points. In Figure 17(c), p_1, p_2 are hashed to the same bucket and p_3 is hashed to a different bucket. Each bucket corresponds to a cluster, thus, we have two clusters in Figure 17(c). Given n gate nodes in the circuit, mapping a solution to Hamming space takes $O(n)$ time, and dimension reduction takes $O(n)$ time. Thus, for m solutions, the total time complexity for clustering is $O(mn)$. Using LSH, one can compute clusters in linear time in terms of m and n.

E. Experimental Results

The new discrete gate sizing algorithm, denoted by NEW, is implemented in C++ and tested on an X86 computer. Our test cases are ISCAS'85 benchmark circuits with a dense $90nm$ gate library where each type of gate has 10 sizes. They are $1\times, 2\times, 3\times, 4\times, 6\times, 8\times, 12\times, 16\times, 24\times, 32\times$ of the minimum size with respect to each cell type.

To judge the efficacy of our discretization algorithm we compare its results with the solutions obtained by simply rounding each size in the continuous solution to the nearest discrete size. In addition, Coudert's approach [6], which is a well-known discrete gate sizing technique, is also implemented for comparison. In this work, we use total capacitance of gates as our area cost function. Comparison results are summarized in Table XII. We make the following observations:

- Nearest rounding always introduces large timing violations. Although the total gate area of nearest rounding is similar to that of continuous solution, its timing is much worse compared to the continuous solution.

- NEW archives much better timing compared to nearest rounding, which makes sense as NEW benefits much from the knowledge of criticality and other global information which is lacking in the case of nearest rounding.

- Compared to [6], 1% - 21% area cost reductions are obtained by NEW.
- Runtime of NEW including computing the continuous solution is on average about 50% higher than [6]. This is already very good considering a dynamic programming-style approach is performed. The efficiency comes from our continuous solution guided scheme and pruning techniques.

Table XII. Comparisons using a library with 10 sizes per gate type. Timing constraints and slack are in *ps*. CPU in seconds is runtime. Area refers to area cost. Area red. refers to the area reduction ratio between NEW and [6].

Circuit	Timing	Continuous Solution			Nearest Round		Approach in [6]			NEW			Area
	Const.	Slack	Area	CPU	Slack	Area	Slack	Area	CPU	Slack	Area	CPU	red.
C432	800	0	4.1	4.1	-42	4.1	12	5.1	10.7	6	4.4	13.7	14%
C499	900	0	7.6	5.9	-94	7.4	12	8.7	28.9	11	8.3	47.8	5%
C880	700	0	5.9	4.2	-115	5.8	4	8.2	21.2	5	6.5	41.7	21%
C1355	1200	0	7.7	7.2	-93	7.5	15	10.1	32.1	26	9.2	37.2	9%
C1908	1200	0	19.7	12.6	-94	19.7	5	24.6	67.5	20	21.7	81.0	12%
C2670	1200	0	20.9	27.8	-102	20.9	37	28.7	121.5	26	23.9	202.1	17%
C3540	1700	0	33.3	21.5	-172	33.2	32	44.4	179.2	41	38.8	284.5	13%
C5315	1500	0	40.1	37.8	-139	40.0	5	55.3	301.8	1	45.7	634.7	17%
C6288	2500	0	33.1	37.6	-191	33.0	19	41.3	403.2	40	38.2	736.0	8%
C7552	2100	0	58.1	67.7	-130	58.1	31	59.7	497.5	1	59.1	534.7	1%

Table XIII. Comparisons using a sparser library with 6 sizes per gate type. Timing constraints and slack are in *ps*. CPU in seconds is runtime. Area refers to area cost. Area red. refers to the area reduction ratio between NEW and [6].

Circuit	Timing	Continuous Solution			Nearest Round		Approach in [6]			NEW			Area
	Const.	Slack	Area	CPU	Slack	Area	Slack	Area	CPU	Slack	Area	CPU	red.
C432	800	0	4.1	4.1	-101	4.0	43	5.4	10.4	4	4.7	12.4	13%
C499	900	0	7.6	5.9	-94	7.4	20	9.1	27.0	12	8.4	40.7	8%
C880	700	0	5.9	4.2	-115	5.8	5	9.0	18.7	1	6.8	33.5	24%
C1355	1200	0	7.7	7.2	-93	7.5	39	11.3	31.5	6	9.1	35.7	19%
C1908	1200	0	19.7	12.6	-147	19.5	1	25.7	60.4	9	22.4	87.1	13%
C2670	1200	0	20.9	27.8	-117	20.8	57	29.0	115.7	83	24.2	188.0	17%
C3540	1700	0	33.3	21.5	-247	33.1	40	44.9	165.8	11	39.4	247.7	12%
C5315	1500	0	40.1	37.8	-155	39.8	7	55.8	329.8	15	46.4	647.8	17%
C6288	2500	0	33.1	37.6	-312	32.7	72	44.2	417.7	45	40.7	775.1	8%
C7552	2100	0	58.1	67.7	-153	58.1	25	61.5	471.9	3	59.0	537.1	4%

As our approach is proposed for cell library based designs, we also inspect how its effectiveness scales with the discreteness in the library. For this, we select six geometrically spaced sizes $(1\times, 2\times, 4\times, 8\times, 16\times, 32\times)$ for each cell type to form a sparser cell library. Gate sizing using nearest rounding and NEW are performed with this new cell library and the results are summarized in Table XIII. We make the following observations.

Nearest rounding often introduces larger timing violations for our geometrically spaced cell library compared to the original cell library. According to Theorem 1, the rounding error is at most 100% since the factor is 2. This is the case for our results. Nearest rounding obtains the same rounding errors for some circuits because the delay of the critical path does not change. This is not surprising as according to our analysis in Section 1, nearest rounding could even obtain smaller timing violations for sparser cell library compared to the original cell library. For detailed comparison on the solutions by nearest rounding and NEW, the histogram for gate sizes for the whole circuit and the critical path of two circuits are shown in Figure 18 and Figure 19. Nearest rounding blindly tries to remain close to continuous solution, while that may not be the best given the discreteness in the library. NEW, since it covers much wider optimization space and is aware of gate criticality, can judiciously upsize gates. For example, it can be seen from Figure 18 that NEW moves a number of gates from size $1\times$ to $2\times$ for the whole design, but on the critical path, it moves significantly more percentage of gates to upper sizes, thus showing the benefit over blind rounding to the nearest size. From Table XIII, one can also see that NEW and [6] are able to obtain the solutions satisfying the timing constraints and the solutions are generally worse than the solutions computed with the original cell library. Compared to [6], 4% - 24% area cost reductions are obtained by NEW. Clearly, our algorithm becomes increasingly useful with sparser cell library.

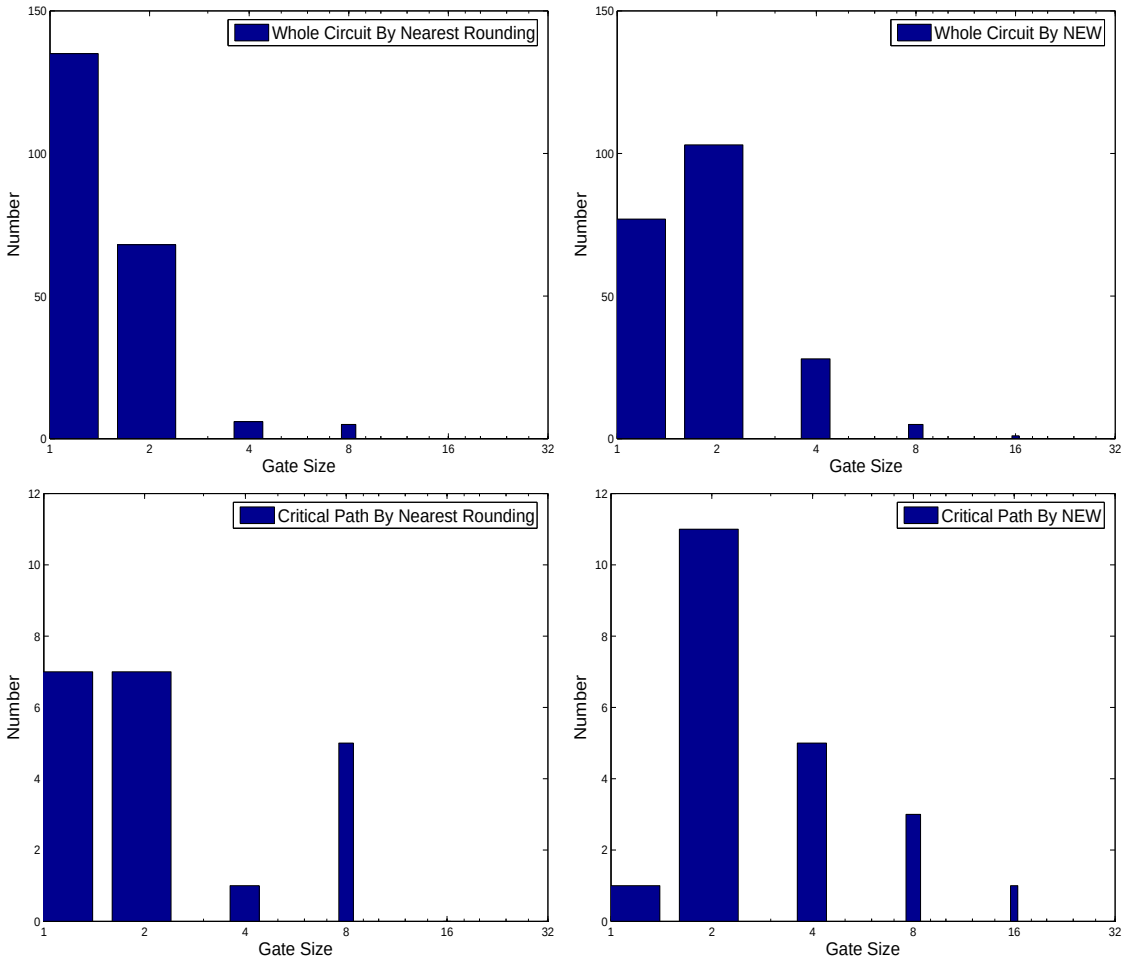


Fig. 18. Gate size histogram for the whole circuit and the critical path of C432 benchmark circuit.

As a byproduct, the proposed algorithm enables us to compute a local delay-area tradeoff curve around the continuous solution. Refer to Figure 20 for two resulting curves. For each plot in Figure 20, the delay-area tradeoff curve computed by NEW is shown and the result by Coudert's approach [6] is also shown for reference. The obtained local tradeoff curve can help users get better timing constraint for the circuit. With new timing constraint, users can use NEW to generate better solutions.

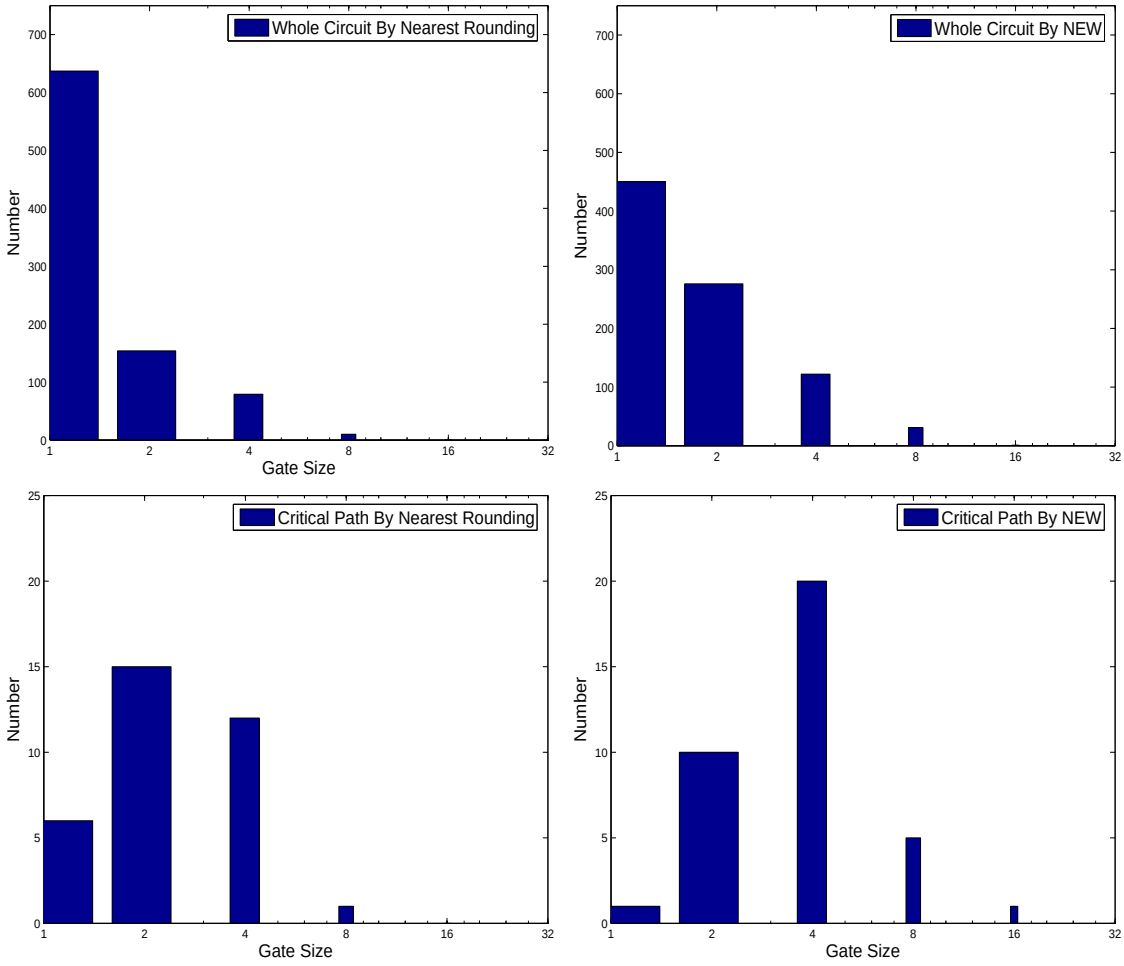


Fig. 19. Gate size histogram for the whole circuit and the critical path of C1908 benchmark circuit.

F. Conclusion

This work proposes a new gate sizing approach which handles the discrete gate library for sparse cell libraries. The new algorithm is based on the idea of continuous solution guided dynamic programming and uses pruning techniques for speedup. Our experimental results demonstrate that up to 21% area cost reduction can be obtained compared to the well-known Coudert's approach. Furthermore, by our approach, a set of trade-offs instead of a single solution are obtained which can help users get bet-

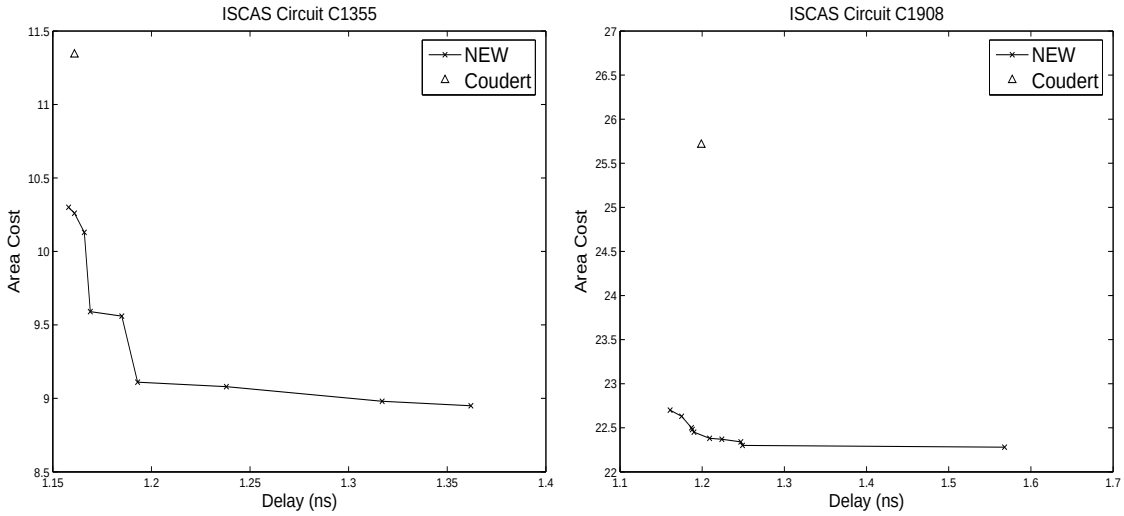


Fig. 20. Delay-cost tradeoff curves for optimizing two ISCAS benchmark circuits. The results of NEW and Coudert's approach [6] are shown.

ter timing constraint for the circuit and provide significant freedom to meet design specifications.

CHAPTER IV

PATTERN SENSITIVE PLACEMENT FOR MANUFACTURABILITY

When VLSI technology scales toward $45nm$, the lithography wavelength stays at $193nm$. This large gap results in strong refractive effects in lithography. Consequently, it is a huge challenge to reliably print layout features on wafers and the printing is more susceptible to lithographic process variations. Although resolution enhancement techniques can mitigate this manufacturability problem, their capabilities are overstretched by the continuous shrinking of VLSI feature size. On the other hand, the quality and robustness of lithography directly depend on layout patterns. Therefore, it becomes imperative to consider the manufacturability issue during layout design such that the burden of lithography process can be alleviated.

In this work, the problem of cell placement considering manufacturability is studied. Instead of designing a new cell placer, our goal is to tune any existing cell placement solution to be lithography friendly. For this purpose, three algorithms are proposed, which are cell flipping algorithm, single row optimization approach and multiple row optimization approach. These algorithms are based on dynamic programming and graph theoretic approaches, and can provide different tradeoff between critical dimension (CD) variation reduction and wirelength increase. Using lithography simulations, our experimental results on realistic netlists and cell library demonstrate that over 15% CD variation reduction can be obtained in post-OPC stage by the new approaches while only less than 1% additional wire is introduced.

A. Introduction

As VLSI technology enters the nano-scale regime, demands for minimum feature sizes have outpaced the advances in lithography hardware solutions. This imposes

a great challenge on manufacturing reliability. In current lithography technology, $193nm$ wavelength is used to print $65nm$ or even $45nm$ features. This leads to a large amount of refractive effects and images on wafer have remarkable mismatches from mask layouts. Lithography-induced variation also aggravates. As more variations are presented with e.g., gate length, timing and power of circuits are significantly affected.

Currently, semiconductor industry heavily relies on *resolution enhancement techniques* (RETs) for improving printability. Roughly speaking, printability refers to the difficulty in obtaining a good match between the intended image and the printed image in lithography process. Printability is often measured by *critical dimension* (CD) accuracy, which refers to the size of thin features (e.g., gate length) which are difficult to print reliably. Thus, achieving high CD accuracy means that the printed patterns well match the desired ones. Prevailing RETs for improving CD accuracy include optimal proximity correction, phase shift mask, off-axis illumination, and sub-resolution assist features [41].

RETs are effective in improving CD accuracy. However, increasingly shrinking features on the die and increasing complexity of the design over-stretch the capability of RETs. This problem aggravates when RETs are applied to the layouts which are not lithography friendly. Furthermore, RETs often complicate photomark shapes and introduce considerable amount of additional cost to photomask fabrication, which makes RETs expensive to apply. To attack the above issues, efforts are needed in all process and design stages. With respect to physical design, manufacturability-aware methodologies would be performed to reduce the burden of manufactures. Our purpose is that with more lithography-friendly layout, the tasks of manufacturers would become significantly easier and RETs become less expensive to apply.

More benefits can be obtained from design for manufacturability. In the sub-

$90nm$ technology, design is heavily affected by fabrication variability. Lithography process certainly has direct impact on fabrication variability. Thus, a lithography-friendly layout has the potential to make the design more resistant to fabrication variations. As the variability has big impact on power, design for manufacturbility also tends to obtain high quality design in terms of power.

There are some previous works related to RET-aware physical designs such as [42, 43] for routing problems. Other lithography friendly design methodologies include regular fabric [44, 45] and restricted design rules (RDR) [46]. Regular fabric methodology, which is somewhat similar to FPGA, requires circuit fabrics to be constructed from a set of regular physical geometry [44, 45]. Due to the geometric regularity, the resulting designs are RET-friendly. However, similar to FPGA based design, circuit performance is compromised. RDR imposes restrictive rules on layout designs to enhance manufacturbility [46]. However, it is difficult and expensive to use these rules to capture the non-local lithography effects. For this, many rules have to be introduced, which may intensify the problem of design rule explosion. Furthermore, RDR introduces regularity into designs which may also lead to penalty on circuit area and performance. In this paper, the problem of manufacturability-driven cell placement problem is studied, and our solutions do not have the above shortcomings. According to the best of authors' knowledge, the closest related work is [47] where a lithography-aware detailed placement approach is presented. It is to achieve high CD accuracy and thus enhance feature printability through perturbing a given detailed placement of circuits. However, it only performs spacing optimization between cells and thus does not allow changing relative locations of cells. Furthermore, it does not consider cell flipping. These turn out to be important to obtain a high-quality lithography-friendly cell placement as indicated by our experiments.

Placement of cells has remarkable effect on printability. This is due to the fact

that gate lengths for transistors on the boundary regions of a cell significantly depend on its neighboring cells. Although sound library cell design can achieve high printability for internal transistors, it cannot handle the boundary transistors. On the other hand, as the gate length keeps shrinking with technologies, the placement will affect deeper and deeper regions of the cells. Since changes on placement may impact the printability on wire, it is suggested to apply our technique for improving printability on placement first, and then printability optimization for wire, local connect and contact to wire can be performed to further improve the printability of the circuit.

In this work, several manufacturability-driven new cell placement algorithms are proposed. Our goal is to modify any existing cell placement solution to make it lithography friendly. The new methods start with a placement obtained from any existing placer, and improve CD accuracy through postprocessing optimizations. Precisely, the location and the orientation of each cell can be perturbed to achieve a lithography-friendly design. As the initial placement is computed by salient (non-lithography-driven) CAD tools and thus of high quality (in terms of e.g., wirelength), it is desired to limit the perturbation to it when improving CD accuracy. Thus, our problem is to compute a lithography-friendly layout subject to perturbation constraints. For this purpose, three algorithms are proposed which are dynamic programming based cell flipping algorithm, single row based optimization approach and multiple row based optimization approach. To measure printability, CD variation is used and it is computed using Calibre LFD which considers OPC effect. To measure perturbation, wirelength increase is used. Our experimental results demonstrate that over 15% CD variation reduction can be obtained in post-OPC stage by the new approaches while only less than 1% additional wire is needed.

The rest of the chapter is organized as follows: Section B formulates the pertur-

bation constrained lithography-driven cell placement problem. Section C describes the cell flipping algorithm. Section D describes single row optimization approach and multiple row optimization approach. Section E presents the experimental results with analysis. A summary of work is given in Section G.

B. Preliminaries

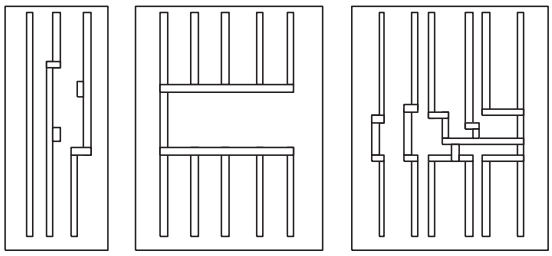
1. Motivation

As demands for minimum feature sizes have outpaced the advances in lithography hardware solutions, there may be considerable amount of distortions between the intended image and the actual printed image in the lithography process. Printability refers to the different levels of distortions. Given an initial cell placement, our goal is to achieve high printability through performing postprocessing optimizations to it.

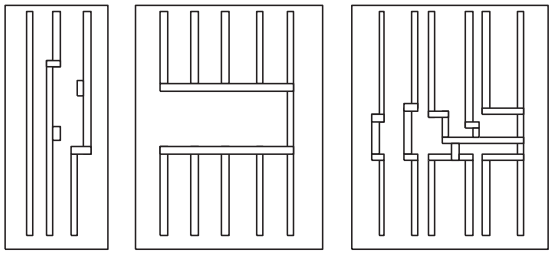
It is helpful to see a simple example which demonstrates that cell placements can affect the printability. Refer to Figure 21 for such an example. Figure 21(a) shows a placement of three gates obtained using a $65nm$ cell library. Figure 21(b) shows another placement obtained by flipping the middle inverter. In this way, we immediately obtain a much more lithography-friendly layout, where CD variation is reduced by $1 - 0.069/0.094 = 27\%$. Note that the above CD variations are obtained from Calibre LFD (Lithography-Friendly Design) which considers OPC effect.

2. Pattern

Distortions in lithography process can be measured by a generic function *pattern dependent manufacturability cost*, or *manufacturability cost* in short, denoted by $\eta(\cdot)$. $\eta(\cdot)$ is defined on *pattern* which is associated with cells, i.e., a pattern could be cells or part of cells. In this paper, we are interested in the pattern spanning only two



(a)



(b)

Fig. 21. Lithography optimization through cell flipping. This design is extracted from an ISCAS'89 benchmark circuit, where an NAND, an inverter and an XNOR gate are placed in series. Though flipping the middle inverter, average CD variation for boundary gate polys is reduced from 9.4% of the nominal value to 6.9% of the nominal value. Rectangles shown are polys. CD variation is obtained from Calibre LFD which considers OPC effects.

horizontally adjacent cells. Denote by C a cell, by C^l the left side of C, and by C^r the right side of C. If a pair of cells C_i, C_j are adjacently placed in a row, a pattern $P(C_i^r, C_j^l)$ associated to them could refer to the part spanning over C_i^r and C_j^l. For each pattern P, we have a manufacturability cost $\eta(P)$. This manufacturability cost can refer to many instances such as CD variations, edge placement error (EPE), image log slope (ILS), process window, or combination of the above [41]. It is important to note the following facts:

- As a cell is associated with an orientation, when the orientation is changed (i.e., cell flipping happens), any pattern associated to this cell is in general also

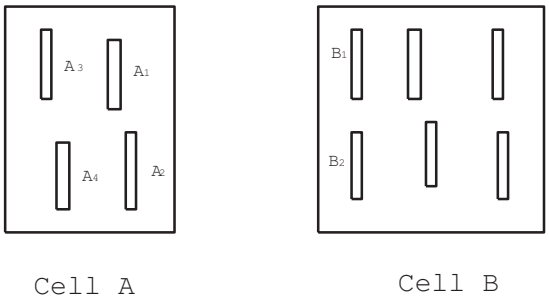


Fig. 22. Definition of manufacturability cost for cells. Rectangles shown are polys.

changed. For example, in Figure 22, η for the pattern between A, B depends on A_1, A_2, B_1, B_2. When cell A is flipped, η for the pattern between A, B will depend on A_3, A_4, B_1, B_2.

- In lithography process, adjacent cells with different spacing can have different printability. This is automatically handled using patterns as the same cell pair with different spacing will be treated as different patterns.

- Since the printability of a cell very weakly depends on cells at other rows [41], it is safe to neglect it. Formally, $\eta(P) = 0$ for any P associated with cells in different rows. This is why we are only interested in pattern associated with horizontally adjacent cells.

Given an initial cell placement, our goal is to reduce the manufacturability cost η through postprocessing optimizations. For this, all patterns in the placement are investigated, and those which are not lithography-friendly, i.e., tend to reduce functional and parametric yield, are identified. Optimizations are then performed there to make the layout lithography-friendly. Thus, our lithography-driven optimization is *pattern sensitive*.

3. Lookup Table for Manufacturability Cost

Online evaluation of manufacturability cost for each pattern is time consuming and not necessary. A better idea is to compute $\eta(P)$ for each possible pattern P off-line and store them in a lookup table for future usage. As P is associated with adjacent cells, thus cell orientations and spacing between cells need to be considered when building the lookup table. The following benefits can be obtained due to lookup table bases lithography optimizations:

- Online lithography simulations, which are very computationally expensive, are avoided. This is a key difference between our approach and those in [42, 43]. There exists fast lithography simulations (e.g., the one in [43]). However, as many runs of online simulations have to be performed during circuit optimizations, such approaches can still be improved. As performing a simulation is much slower than obtaining a number in a lookup table, one could use lookup table based optimization for speedup.

- As the lookup table is built off-line, full-fledged expensive lithography simulations can be performed. Compared to [43] where a fast aerial image (which is a first order approximation to the optimal system) simulation is performed, we estimate printability using more accurate approximations.

- OPC can be performed to the circuit pattern before computing the manufacturability cost of it. It is well known that OPC effects are very difficult to model, however, our lookup table based methodology can easily handle them. Note that other RETs can also be applied before computing the manufacturability cost for each pattern.

One may wonder whether it is practical to use lookup table to characterize all boundary patterns for a large cell library. Note that our technique should be applied mainly to a local region (i.e., critical part) of circuits instead of the whole circuit. The cell library can be then largely reduced. In addition, two techniques can be applied to further reduce the size of the lookup table. First, it is interesting to see that boundary patterns can be similar for different cell pairs. For example, in Figure 21, the left boundary polys of the NAND and the inverter are the same. Although cell library can be large, the number of dramatically different boundary patterns could be limited. Thus, one could characterize some representative patterns using lookup table and perform fuzzy matching based search (which is similar to the approach in [48]) in optimization. Second, one could specify a number of candidate boundary poly shapes, and then force the boundary poly (of each cell type) to be one of the candidate shapes in designing the cell library. In this way, the number of different boundary patterns can also be significantly reduced [49].

4. Problem Formulation

As manufacturability cost between cells in different rows is negligible, row-based placement approaches are designed in this paper. For this, we define *row η cost* for a row of cells as the sum of η between all adjacently placed cell pairs in the row, and define the *total η cost* for the whole placement as the sum of all row η costs. In this paper, we propose to adjust the placement to reduce the total η cost. Such adjustment should be as slight as possible so as to introduce minimal amount of perturbation to the design. This is desired as the initial design, although not lithography-friendly, should have high quality in terms of e.g., wirelength as it is returned by salient (non-lithography-driven) CAD tools.

Three types of adjustments are considered in this paper. The first type of ad-

justments is not to change the locations of cells, rather, it only allows changing orientations of cells. We call it *Cell Flipping Optimization*. Refer to Figure 21 for an example illustrating the impact of cell flipping on η. Evidently, the manufacturability cost is significantly reduced in this example. Note that cell flipping optimization has also been used in [50] for wirelength reduction in cell placement. The second type of adjustments allows both cell re-location and cell flipping. To introduce small amount of perturbation to the original placement, each cell is only allowed to move within a small range around its original location. Furthermore, when a row is adjusted, all other rows must be fixed. Thus, this type of adjustments is called *Single Row Optimization*. The third type of adjustments is the same as single row optimization except that several neighboring rows are optimized simultaneously, that is, when a row is adjusted, it neighboring rows are adjusted as well. This type of adjustments is called *Multiple Row Optimization*. Clearly, increasing amount of efforts are needed in these three optimizations, and one may expect that increasing amount of reduction in manufacturability cost can be obtained. Our problem is formulated as follows.

Perturbation Constrained Lithography-Driven Cell Placement Problem: Given a cell placement, we are to perform post-processing optimizations, which can be cell flipping, single row optimizations or multiple row optimizations, such that the total manufacturability cost (i.e., total η cost) is reduced subject to the constraint α on perturbation.

In this paper, we measure the *perturbation* to a placement by wirelength increase as wirelength has direct relationship with timing and is efficient to compute. Thus, the perturbation constraint α actually refers to the maximum tolerable wirelength increase ratio. Note that other similar metrics could be easily incorporated into our new approach.

Finally, although this paper is restricted to row-based postplacement designs, the ideas can be extended to handle non-row-based designs. In particular, Multiple Row Optimization approach can be directly applied to other placement styles.

C. Cell Flipping

The first algorithm, namely, cell flipping algorithm, works under the dynamic programming framework. As our cell placement algorithm is a row-based approach, cell flipping algorithm is carried out row by row. In a row, the location of each cell is fixed and the orientation of each cell is to be determined. For convenience, "cells" in this section simply refer to a row of cells.

We define a *partial* cell flipping solution to be an incomplete determination for the orientations of all cells. A partial solution becomes *complete* when the orientations of all cells are determined. A cell is *processed* if its orientation has been determined.

1. Algorithmic Overview

Given an initial cell placement, cells are first sorted in the topological order (i.e., from left to right). We then start from the first cell and set its orientation to each of two possible choices (i.e., flipped or un-flipped), which results in two partial solutions. For each partial solution, we process the second cell and set its orientation to each of two choices. In this way, the algorithm proceeds in a dynamic programming fashion, i.e., it processes each cell in turn according to the topological order. Without any solution pruning, there are certainly 2^n solutions for optimizing n cells. Therefore, during the solution propagation process, inferior solutions are pruned for acceleration. The algorithm terminates when all partial solutions become complete and the solution with the minimal η cost and satisfying wirelength constraint is returned.

2. Solution Characterization

A set of partial solutions $\mathcal{S}$ keep being updated during the process of dynamic programming. Each solution $S \in \mathcal{S}$ is associated with a (CE, CW) pair, where CE denotes the cumulative η cost for all processed cells, and CW denotes the cumulative wirelength. Note that CW is computed using the widely-used metric *half-perimeter wirelength* (HPWL) on all those nets which do not span on any unprocessed cell.

3. Solution Propagation

Suppose that a cell C is "inserted" to the current partial solution S, i.e., we are to decide the orientation of C. A new solution S' will be formed for each possible cell insertion (flipped C and unflipped C). Because all cells are processed according to the topological order, when C is processed, the cumulative η cost can be updated by

$$CE(S') = CE(S) + \eta(P(C^r_{last}, C^l)),$$

where C_{last} is the last processed cell. The cumulative wirelength is updated by re-computing HPWL of all nets not spanning on any unprocessed cell.

4. Solution Pruning

During the process of dynamic programming, there may be a lot of solutions. Some of them are inferior to others and they will be pruned to accelerate the approach.

For any two solutions S_1, S_2 *with the same set of processed cells*, we say that S_2 is *inferior to* S_1 if the following two conditions are satisfied. First, the cumulative wirelengths are compared and we need that $CW(S_2) \geq CW(S_1)$. Second, to compare manufacturability costs, one has to consider the effect due to the next cell. Denote by $\eta_{next}(S)$ the manufacturabiliy cost for the pattern formed by the current cell and

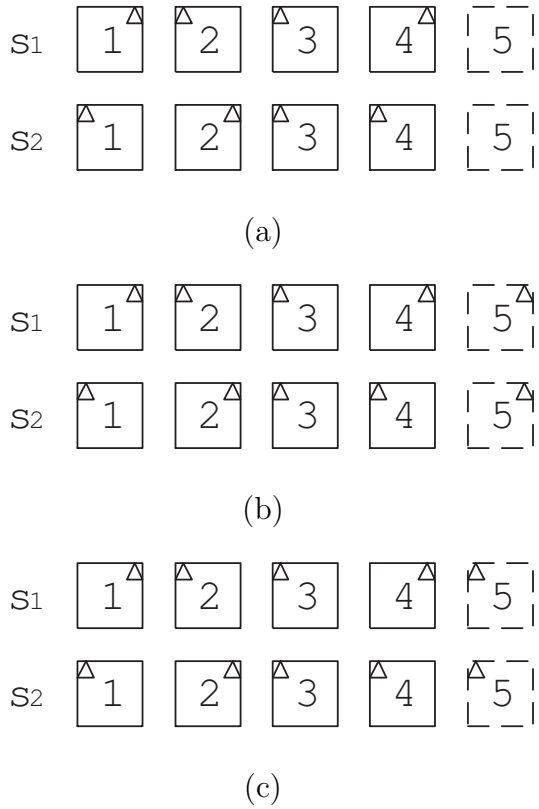


Fig. 23. Solution pruning: (a) before pruning (b) inferiority check when the fifth cell is unflipped (c) inferiority check when the fifth cell is flipped. The triangle denotes the cell orientation. A cell with triangle on the right denotes an unflipped cell and with triangle on the left denotes a flipped cell.

the next cell (if available). Thus $\eta_{next} = \eta(P(C^r, C^l_{next}))$ where C_{next} denotes the next cell.

Refer to Figure 23. Figure 23(a) shows two solutions where three cells have been processed and the fourth cell is being processed. In order to decide which solution has the better manufacturability cost, the fifth cell is first set to be unflipped and then set to be flipped. In both scenarios, if $CE(S_2) + \eta_{next}(S_2) \geq CE(S_1) + \eta_{next}(S_1)$ consistently happens, we can say that S_1 must be at least as good as S_2 in terms of manufacturability cost. The reason is that given a complete solution achieving the best manufacturability cost constructed from S_2, one can always replace S_2 with S_1

to get a complete solution with no more manufacturability cost.

Note that the above pruning technique is better than the one in [51]. In [51], when comparing the manufacturability costs of S_1 and S_2, the pattern between the fourth cell and the fifth cell is not considered. That is, [51] only needs $CE(S_2) \geq CE(S_1)$ for comparison on manufacturability cost. Since the orientation of the fourth cell certainly has impact on the following cell (i.e., the fifth cell), the technique in [51] could prune potentially better solutions terms of manufacturability. The experiment in Section E demonstrates this.

In summary, a solution is inferior to another if it has worse cumulative η cost and worse cumulative wirelength. Whenever a solution becomes inferior, it is pruned from the solution set without further propagation. A solution S can also be pruned when it is infeasible, i.e., its cumulative wirelength is greater than the wirelength constraint of that row. For a row, the wirelength constraint is set to $(1+\alpha) \cdot L$, where L is the total wirelength of the row in the original (i.e., initial) placement and α is the maximum tolerable wirelength increase ratio.

The pseudocodes for cell flipping algorithm is shown in Figure 24.

Algorithm: Cell flipping for a single row.
Input: $\mathcal{C}$: cells to be placed, an initial cell placement, α: total wirelength constraint **Output:** cell placement with reduced η satisfying α
1. Topologically (i.e., from left to right) sort all cells in $\mathcal{C}$ in the initial placement 2. $\mathcal{S} = \emptyset$ 3. for each cell C in the topological order, do 4. for each solution S in $\mathcal{S}$, do 5. for each of two possible orientations, do 6. // generate a new solution S' 7. compute $CE(S')$ and $CW(S')$ 8. insert S' into $\mathcal{S}$ and perform pruning if necessary 9. return S with the minimal η cost and satisfying α

Fig. 24. Cell flipping algorithm for a single row.

D. Single Row Optimization and Multiple Row Optimization

1. Algorithmic Overview (Single Row Optimization)

In single row based optimization, in addition to the cell orientation, we are allowed to change the location of each cell. In this way, more manufacturability cost (i.e., η cost) reduction is expected. Since small perturbation is desired, each cell is only allowed to be movable within a small range around its original position. To approximately implement this strategy, cells will be processed by groups. Every consecutive k cells form a *group* and optimizations (including relocation and cell flipping) are performed inside each group. At any time, only one group is optimized. To determine which group to be optimized, a multi-dimensional descent based optimization approach is used. Such an approach has been successfully used in CAD problems including [6] for gate sizing.

At the beginning, a set of groups called *improvablegroups* are formed as follows. Each group in the row will be assigned with a *cost* which is equal to the possible η reduction for this group (computed by tentatively optimizing the group) when all other cells are fixed. As long as the cost for a group is positive (i.e., η reduction is possible), the group is included into improvablegroups. We then compute the ratios of η reduction over wirelength increase for all groups in improvablegroups. A subset of improvablegroups, called *optimizedgroups*, are then computed as an ordered set of groups with ratios greater than a threshold as they may provide cumulative improvement in cost with small wirelength increase. Optimizations are then performed to each group in optimizedgroups in turn. Subsequently, improvablegroups are set to optimizedgroups and the above process is repeated until convergence.

The remaining question is how to compute η reduction for a group of cells. Precisely, our goal is to compute a new cell placement (for this group of cells) with

reduced η cost subject to the constraint on wirelength increase. For this, we will first compute the "best η" solution, i.e., the optimal η-driven placement solution without considering wirelength constraint. Since the original placement is returned by salient CAD tools, it is reasonable to treat it as the "best wire" solution. Subsequently, an iterative approach is performed to gradually turn the best η solution into the best wire solution. The process terminates when wirelength increase ratio satisfies the constraint α (together with non-overlapping requirement). Since our goal is to obtain a minimum η solution subject to the wirelength constraint, it makes sense to gradually modify the best η to obtain the solution satisfying the wirelength constraint and still with good η. The approach will be detailed in Section 2 and Section 3.

2. Unconstrained Optimal Manufacturability-Driven Placement

A critical observation is that when wirelength is not considered, for a group of cells, the cell placement achieving the minimal η cost can be obtained through reduction to the *minimum cost Hamiltonian path problem*.

A graph $G = (V, E)$ is to be constructed as follows. Each cell C in the group will be mapped to two nodes $v_{C,l}$ and $v_{C,r}$ in G, where $v_{C,l}$ corresponds to the left side of unflipped C and $v_{C,r}$ corresponds to the right side of unflipped C. There is an edge between $v_{C,l}$ and $v_{C,r}$ with weight 0. For any two nodes v_i and v_j which belong to different cells, there is an edge (v_i, v_j). The weight associated with such an edge is equal to the smallest η cost between v_i and v_j when placing v_i (on the left) and v_j (on the right) adjacently, i.e., $P(v_i, v_j)$. Note that η cost function is in general not monotonic with whitespace [47], and the smallest η can be obtained by our η lookup table. Refer to Figure 26 for the graph corresponding to Figure 25 which is a placement with three cells. Note that weights for edges $(v_{A,l}, v_{A,r}), (v_{B,l}, v_{B,r}), (v_{C,l}, v_{C,r})$ are all 0. As an example, we show how to compute the weight of edge $(v_{A,l}, v_{B,r})$.

Since the weight of an edge refers to the η for the part between the two nodes, we have to first flip cell A and cell B to make the left side of cell A (corresponding to $v_{A,l}$) directly connect to the right side of cell B (corresponding to $v_{B,r}$). Denote the flipped C by C_f. Thus, the weight for $(v_{A,l}, v_{B,r})$ is equal to $\eta(P(C^l_{A,f}, C^r_{B,f}))$.

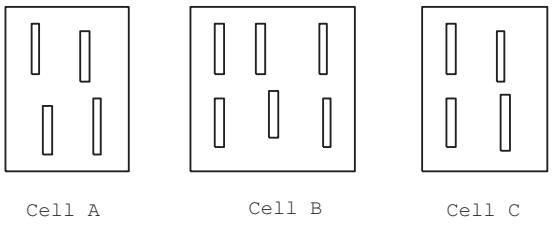


Fig. 25. A placement with three cells.

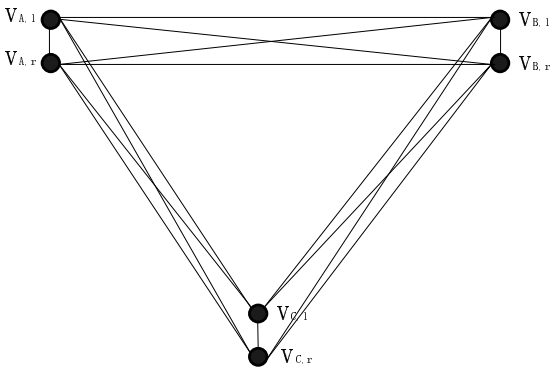


Fig. 26. Graph G corresponding to Figure 25.

To compute the optimal η-driven cell placement (i.e., best η solution) for this group of cells, it suffices to compute a path visiting each node exactly once such that the total edge weights along the path is minimized. This problem is the *Minimum Cost Hamiltonian Path Problem*. For example, in Figure 26, if $v_{B,l}v_{B,r}v_{A,r}v_{A,l}v_{C,l}v_{C,r}$ is returned as the minimum cost Hamiltonian path, then we know that in best η solution, we need to place B, A, C in this order and B, C are unflipped while A is flipped.

As the minimum cost Hamiltonian path problem is an NP-complete problem [52], the following closest-point heuristic (which is similar to the one in [52]) is used to compute the efficient approximation. Define *a partial Hamiltonian path* to be an incomplete Hamiltonian path. The algorithm begins with picking an arbitrary node in G. At each step, the node which is not yet included in the partial Hamiltonian path and is closest to any point along the partial Hamiltonian path is identified. Denote this node by u and suppose that it is closest to v along the partial path. We will insert u to the path just after v. Note that whenever a node is included, another node belonging to the same cell (i.e., the node corresponding to another side of the cell) must also be included. For example, if $v_{A,r}$ is picked for insertion, then $v_{A,l}$ must be inserted immediately after $v_{A,r}$. In this way, we guarantee that the resulting path is valid for placement. It is clear that the close-point heuristic has this property.

3. Manufacturability-Wirelength Tradeoff

When wirelength is not considered, optimal η-driven placement can be computed as in Section 2. This subsection deals with turning an unconstrained solution into a wirelength constrained solution. Our idea is to start from the best η solution and perform local adjustment to make its wirelength increase fall into the requirement. Precisely, our local adjustment is an iterative approach and it gradually turns the cell placement to be closer and closer to the original placement until the wirelength increase satisfies the constraint. Since η cost and wirelength depend on the spacing (i.e., whitespace) between cells, our approach also performs spacing optimization.

Let us illustrate the approach using a simple example. Suppose that the original cell placement and optimal η-driven cell placement are as shown in Figure 27. Our algorithm first identifies an ordered set of cell pairs for location exchange to turn the optimal η-driven placement (best η solution) into the original cell placement (best

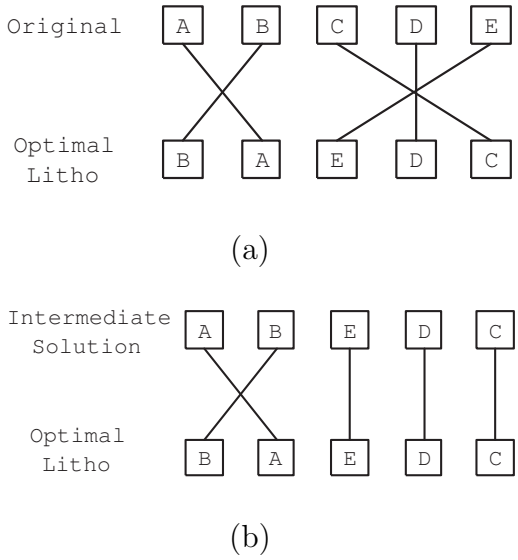


Fig. 27. Obtaining tradeoff between manufacturability cost and wirelength: (a) two initial solutions with best manufacturabiliy cost (Optimal Litho) and best wirelength (Original) (b) an intermediate solution is obtained by exchanging cells with maximum link crossings (which are C, E in this case).

wire solution). For this, an iterative procedure is used. At each iteration, a pair of cells are identified for location exchange. A cell is first linked to its target location (which is the location in best wire placement) as in Figure 27. A cell which is not at its target location and is with maximum link crossings is then identified[1]. Another cell to be exchanged with it is the one at the target location of the identified cell. For example, in Figure 27, cell E is first identified and it is to exchange location with cell C. It is easy to see that the ordered set of exchange cell pairs are $\{EC, BA\}$.

After identifying the ordered set of cell pairs, we are to perform cell location exchange one by one. After each location exchange, spacings between cells are inherited from the ones before location exchange with the following exception. For those placement patterns of cell pairs which can be found in the best wire placement, spac-

[1]In case of a tie, an arbitrary cell in tie is identified.

ings between them are set as in the best wire placement. For example, after E, C are exchanged in Figure 27, spacing between CD and DE will be set as in original placement and spacing between AC is inherited, i.e., it is equal to that of AE in the optimal η solution. Note that since our optimizations are performed within a group, we need to guarantee that after optimization, placement of the group will not overlap with its neighboring groups. Thus, location exchange process terminates when wirelength increase ratio of the placement falls below α and the placement does not overlap with its neighboring groups. In this way, we can obtain a good η-driven cell placement subject to the wirelength constraint.

The pseudocodes for Single Row Optimization is shown in Figure 28 and Figure 29.

Algorithm: Single row optimization.
Input: an initial cell placement and α **Output:** η-driven cell placement satisfying α
1. for each row in the topological order, do 2. improvablegroups ← groups with positive costs using Group Optimization 3. repeat 4. compute optimizedgroups as a subset of improvablegroup which can provide cumulative improvement in cost 5. perform optimizations to optimizedgroups using Group Optimization 6. cost re-evaluation for optimizedgroups using Group Optimization 7. improvablegroups ← optimizedgroups 8. until convergence 9. choose cell placement with minimum η cost

Fig. 28. Single row optimization algorithm.

Algorithm: Group optimization.
Input: an initial cell placement and α **Output:** η-driven cell placement satisfying α
1. form a graph corresponding to the group of cells 2. compute optimal η-driven placement through reduction to min-cost Hamiltonian path problem. 3. identify an ordered set of cell pairs for location exchange 4. for each pair in the set, do 5. perform location exchange 6. break if wire increase ratio satisfies α and non-overlapping requirement 7. return the feasible cell placement for the group

Fig. 29. Group optimization algorithm.

4. Extension to Multiple Row Optimization

Suppose that there are five rows of cells to be placed. By single row optimization, these five rows will be optimized separately, i.e., the first row of cells will be optimized followed by the second row and so forth. However, when a row is optimized, it would be possible that its neighboring rows need to be accordingly modified to achieve an overall good design. This is due to the fact that a net often spans a few neighboring rows, and thus wirelength could be reduced with adjusting several rows simultaneously. As a consequence, some previously "infeasible" placements (i.e., the one violating wirelength constraint) which provide large amount of η reduction may become feasible. This may eventually leads to that the solution with more η reduction is returned. Refer to Figure 30 for an example. Figure 30(a) shows the original circuit. During single row optimization for the third row of cells, we may obtain an intermediate solution as shown in Figure 30(b). There may be too much wirelength increase if we further perform optimizations in this row and then the solution will be pruned. However, if cells in other rows can be moved at this point, we may get Figure 30(c) which is good in both wirelength and manutfactruability cost. This

motivates multiple row optimization.

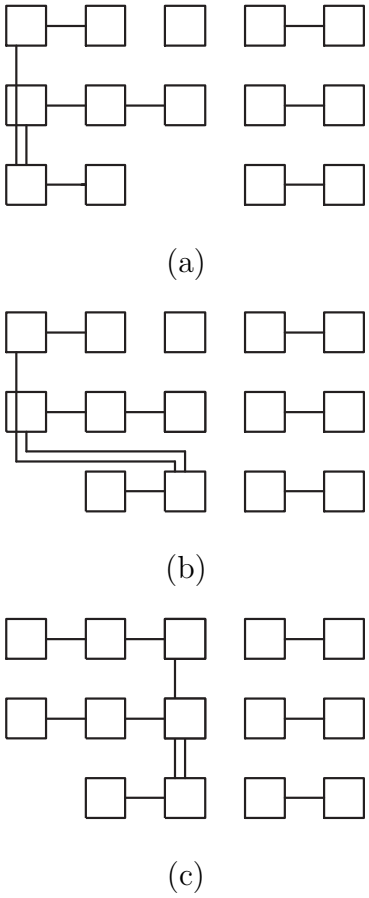


Fig. 30. An example of multiple row optimization: (a) original circuit (b) an intermediate solution in single row optimization (c) an intermediate solution in multiple row optimization.

The algorithm for single row optimization can be readily extended to handle multiple row optimization. In multiple row optimization, m neighboring rows will be grouped to form a *row group* and optimizations are performed within each row group. The algorithm is as follows.

For each row within a row group, cells are grouped as in single row optimization. All resulting groups (in all rows of the row group) are put into a single set called *group set*. Our purpose is to treat groups from different rows in the same way for

"simultaneous optimization". It is true that at any time, only one group can be optimized. However, viewing at the level of group set optimization, groups of different rows can be optimized interactively. For example, it is possible that optimizing a group at a row may impact some groups at other rows and performing successive optimizations there may be very beneficial. Our multiple row optimization captures this and interactions among neighboring rows are explored. In this sense, multiple rows are optimized "simultaneously". Since any group in the group set must be located in a single row, approaches for computing wirelength constrained η-driven placement as described in Section 2 and Section 3 can be directly applied. The multi-dimensional descent based approach is also readily applied. Precisely, we pick a candidate set of groups (which may be in different rows) as improvablegroups and identify a subset of it which can improve cost. After performing optimizations to this subset, improvablegroups is updated and this process is repeated until convergence. Note that since our optimization is performed into a group whose cells must be in the same row, no cell can move to any other row after optimization.

The pseudocodes for the above algorithm is shown in Figure 31 and Figure 29.

E. Experimental Results

1. Experiment Setup

The algorithms of Cell Flipping, Single Row Optimization and Multiple Row Optimization are implemented in C++ and are tested on a Pentium IV computer with a 2.8GHz CPU. Two sets of testcases are used in experiments, namely, ISPD'04 benchmark circuits and ISCAS'89 benchmark circuits. In performing lithography-driven postprocessing optimizations, the perturbation constraint, i.e., the maximum tolerable wirelength increase ratio α for the circuit, is set to 1% as an example. Other

Algorithm: Multiple row optimization.
Input: an initial cell placement and α **Output:** η-driven cell placement satisfying α
1. group neighboring rows to form row groups 2. for each row group, do 3. improvablegroups ← groups of all rows with positive costs using Group Optimization 4. repeat 5. compute optimizedgroups as a subset of improvablegroups 6. perform optimizations to optimizedgroups using Group Optimization 7. cost re-evaluation for optimizedgroups using Group Optimization 8. improvablegroups ← optimizedgroups 9. until convergence 10. choose cell placement with minimum η cost

Fig. 31. Multiple row optimization algorithm.

wirelength increase ratio can be used. Similar to previous work [47], manufacturing distortions are measured by CD variation of boundary gate poly in this paper. Due to the fact that variations on boundary gate polys have the most significant effect on circuit performance, CD variation for boundary gate poly is used. However, we can certainly measure CD variation for all gate polys in a cell. This only changes our lookup table but not the algorithms.

2. Experiments with ISCAS'89 Benchmark Circuits

We first perform experiments on ISCAS'89 benchmark circuits. Logical synthesis and technology mapping (using Berkeley SIS) with a cell library for $65nm$ technology, which consists of 22 cells, are performed to the circuits. Our lookup table for manufacturability cost is built as follows. For each pair of cell types, for each possible cell orientation, for each representative spacing between cells, an CD variation is obtained by Mentor Graphics Calibre LFD, which is a commercial tool for performing

lithography simulations. Thus, CD variations in our lookup table are accurate in contrast to fast estimation on CD variation. RETs are often performed to improve the printability in practice. To capture this, OPC is performed on the pattern using Calibre LFD before computing the manufacturability cost by simulations. Note that other RETs can also be applied. Due to lack of large industrial cell library, our cell library consists of 22 cells and our lookup table size is also not large. Note that our approach is mainly applied to optimize small critical region of circuits which consist of small number of cells. To handle a large cell library, there are many techniques for effectively reducing the lookup table size as described in Section 3 (e.g., we only need to select a few representative patterns to put into lookup table). Note that as cell library characterization for delay, cell library characterization for manufacturability (i.e., the lookup table) is performed offline and can be reused later. Initial placements are computed using FastPlace [53]. Note that other placers can also be used. Placements are then optimized for CD variation reduction, where CD variation reduction is defined as $1 - \frac{\text{CD variation after optimization}}{\text{CD variation before optimization}}$. The results are summarized in Table XIV. We make the following observations.

- For Cell Flipping algorithm, on average about 5% CD variation reduction is obtained with 0.13% additional wire. Cell Flipping algorithm runs fastest among all algorithms, which makes sense as the smallest amount of effort is needed there.

- For Single Row Optimization algorithm, on average 9.8% CD variation reduction is obtained, which improves the results by Cell Flipping. The amount of additional wire is still small. On average, only 0.32% more wire is needed.

- For Multiple Row Optimization algorithm, on average 15.2% CD variation reduction is obtained, which gives the best results among all three algorithms.

At the same time, it needs more wire. Multiple Row Optimization runs slowest among all three algorithms.

Table XIV. Performance of each algorithm on ISCAS'89 benchmark circuits. W.I. refers to the wirelength increase and V.D. refers to the variation reduction. CPU time is in seconds.

Circuit		Cell Flipping			Single Row Optimization			Multiple Row Optimization		
Name	#Nodes	W.I.	V.R.	CPU	W.I.	V.R.	CPU	W.I.	V.R.	CPU
s838	317	0.04%	6.7%	0.1	0.06%	10.5%	0.1	0.31%	16.8%	0.2
s1238	439	0.15%	4.2%	0.2	0.18%	9.8%	0.3	0.34%	15.4%	0.5
s1423	511	0.04%	5.4%	0.2	0.20%	10.7%	0.3	0.45%	15.8%	0.5
s1488	429	0.06%	5.6%	0.2	0.38%	10.0%	0.4	0.61%	14.6%	0.5
s5378	1227	0.19%	6.5%	0.7	0.57%	8.3%	0.8	0.72%	12.3%	1.2
s9234	1162	0.18%	5.0%	0.6	0.29%	9.2%	0.9	0.50%	13.6%	1.3
s15850	3621	0.05%	5.9%	2.9	0.31%	11.2%	5.2	0.39%	17.2%	8.1
s35932	13460	0.17%	4.8%	24.3	0.48%	8.5%	58.0	0.73%	14.3%	103.1
s38417	8965	0.32%	5.3%	11.4	0.59%	9.5%	18.9	0.81%	15.7%	39.3
s38584	10463	0.07%	5.8%	14.8	0.11%	9.9%	22.5	0.32%	16.5%	43.5
Average	4059	0.13%	5.5%	5.5	0.32%	9.8%	10.7	0.52%	15.2%	19.8

Our optimization approaches can be tuned to obtain different tradeoff between CD variation reduction and wirelength. For example, in Single Row Optimization, one can impose a maximum number of iterations to the implementation so as to terminate the multi-dimensional descent based optimization procedure before it converges. By varying this number, different tradeoff can be obtained. As an example, a tradeoff curve is shown in Figure 32.

As mentioned in Section 4, our pruning technique is better than [51] since the new pruning technique considers the impact to the not-yet-processed cells. We also perform experiments to demonstrate this. The results using the pruning technique in [51] are summarized in Table XV. Comparing Table XIV and Table XV, one can see that the the new pruning technique leads to better manufacturability cost than the one in [51].

In Section 3, we propose a crossing reduction technique to achieve the tradeoff between the best manuacturability cost solution and the best wirelength solution. We

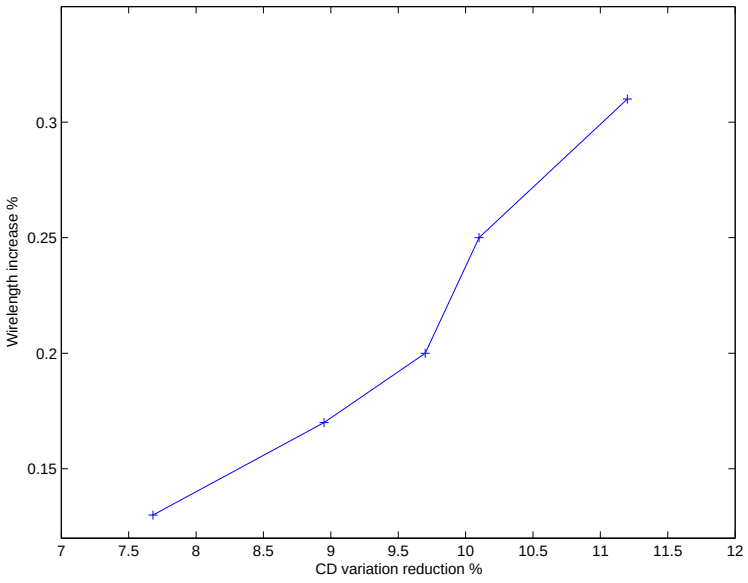


Fig. 32. Tradeoff between CD variation reduction and wirelength increase using single row optimization for s15850.

are to investigate the effectiveness of the tradeoff technique. For this, we turn off the tradeoff option and run the experiments as above. That is, we always set the placement to the best manufacturability cost solution without considering the wirelength constraint. The results are summarized in Table XVI. Comparing Table XIV and Table XVI, one can see that CD variation is further reduced, however, wirelength overhead is significantly increased (2.53% without considering wirelength constraint v.s. 0.32% considering wirelength constraint). Thus, the best manufacturability cost solution introduces too much overhead into the original design and is less useful. Our tradeoff technique proposed in Section 3 is very effective in achieving a good tradeoff between manufacturability cost and wirelength overhead.

3. Experiments with ISPD'04 Benchmark Circuits

We next perform experiments on a standard placement benchmark, namely, ISPD'04 benchmark circuits [53]. The statistics of the circuits are shown in Table XVII. The

Table XV. Cell flipping using pruning technique in [51].

Circuit	Wire Inc.	Variation Red.	CPU (s)
s838	0.03%	4.1%	0.1
s1238	0.10%	3.2%	0.2
s1423	0.04%	5.2%	0.2
s1488	0.07%	5.0%	0.2
s5378	0.15%	5.6%	0.6
s9234	0.12%	3.9%	0.5
s15850	0.05%	5.7%	2.7
s35932	0.22%	3.7%	21.8
s38417	0.30%	4.8%	10.9
s38584	0.05%	5.5%	13.7
Average	0.11%	4.7%	5.1

circuits are first placed to obtain initial placement results using FastPlace [53]. They are then optimized for manufacturability cost and the results are summarized in Table XVIII. Since the original circuits for ISPD'04 benchmark are not known to us, their gate types are randomly assigned from our cell library. We make the following observations.

- For Cell Flipping algorithm, on average more than 5% CD variation reduction is obtained, with only 0.10% additional wire.
- For Single Row Optimization algorithm, on average 11.0% CD variation reduction is obtained, which improves the results by Cell Flipping. The amount of additional wire is still small. On average, only 0.24% more wire is needed.
- For Multiple Row Optimization algorithm, on average 15.5% CD variation reduction is obtained, which gives the best results among all three algorithms. At the same time, it needs 0.35% additional wire. As it spends a lot of effort on optimizations, it runs slowest among all algorithms.

Table XVI. Single Row Optimization results without considering wirelength constraint.

Circuit	Wire Inc.	Variation Red.	CPU (s)
s838	2.85%	19.0%	0.1
s1238	2.47%	16.8%	0.3
s1423	4.52%	15.5%	0.3
s1488	2.17%	17.0%	0.3
s5378	2.96%	14.5%	0.7
s9234	1.82%	13.3%	0.8
s15850	1.51%	18.2%	4.9
s35932	1.28%	10.1%	52.5
s38417	4.32%	16.5%	13.2
s38584	1.35%	12.9%	17.2
Average	2.53%	15.4%	9.0

F. Conclusion

Traditionally, design and manufacturing process are separate. This trend should be turned so as to make the resolution enhancement techniques easy and less expensive to apply. In this paper, several new algorithms are proposed for manufacturability-driven cell placement. They are cell flipping algorithm, single row based optimization and multiple row based optimization approaches. Cell flipping algorithm works under the dynamic programming framework. In row-based optimizations, cells are partitioned into groups and are optimized through reduction to graph theoretic problems such as the minimum cost Hamiltonian path problem. These algorithms are very effective in reducing CD variations and are able to provide different tradeoff between CD variation reduction and wirelength increase. Our experimental results demonstrate that $> 15\%$ reduction in CD variation can be obtained in post-OPC stage by the new approaches with only $< 1\%$ additional wire. Although this paper is restricted to row-based designs, the ideas can be extended to handle non-row-based designs. In particular, Multiple Row Optimization approach can be directly applied to other

Table XVII. Statistics of ISPD'04 benchmark circuits [53].

Circuit	#Cells	#Pads	#Nets	#Pins	#Rows
IBM01	12506	246	14111	50566	96
IBM02	19342	259	19584	81199	109
IBM03	22853	283	27401	93573	121
IBM04	27220	287	31970	105859	136
IBM05	28146	1201	28446	126308	139
IBM06	32332	166	34826	128182	126
IBM07	45639	287	48117	175639	166
IBM08	51023	286	50513	204890	170
IBM09	53110	285	60902	222088	183
IBM10	68685	744	75196	297567	234
IBM11	70152	406	81454	280786	208
IBM12	70439	637	77240	317760	242
IBM13	83709	490	99666	357075	224
IBM14	147088	517	152772	546816	305
IBM15	161187	383	186608	715823	303
IBM16	182980	504	190048	778823	347
IBM17	184752	743	189581	860036	379
IBM18	210341	272	201920	819697	361

placement styles.

Table XVIII. Performance of each algorithm on ISPD'04 benchmark circuits. CPU time is in seconds.

	Cell Flipping			Single Row Optimization			Multiple Row Optimization		
Circuit	Wire Inc.	Var. Red.	CPU	Wire Inc.	Var. Red.	CPU	Wire Inc.	Var. Red.	CPU
IBM01	0.02%	5.3%	15.7	0.30%	12.8%	32.8	0.38%	17.8%	55.7
IBM02	0.10%	5.0%	33.5	0.17%	10.9%	81.7	0.32%	18.5%	139.2
IBM03	0.18%	6.1%	51.0	0.35%	11.2%	127.8	0.39%	14.8%	205.7
IBM04	0.12%	5.9%	65.1	0.42%	10.7%	190.5	0.48%	17.3%	274.5
IBM05	0.05%	7.1%	75.5	0.12%	13.4%	215.7	0.15%	15.6%	350.9
IBM06	0.08%	4.8%	98.3	0.25%	9.5%	298.8	0.37%	13.1%	507.8
IBM07	0.10%	5.4%	182.1	0.16%	11.4%	502.2	0.22%	18.2%	983.5
IBM08	0.13%	7.2%	279.8	0.22%	12.1%	557.6	0.46%	16.9%	1378.4
IBM09	0.14%	5.2%	383.0	0.20%	9.9%	832.0	0.50%	12.7%	1518.6
IBM10	0.11%	6.3%	489.2	0.17%	12.3%	1013.5	0.21%	17.1%	1591.2
IBM11	0.20%	6.5%	675.3	0.38%	11.6%	1295.8	0.54%	15.4%	1765.7
IBM12	0.12%	5.6%	820.2	0.27%	10.8%	1775.7	0.31%	12.9%	2031.8
IBM13	0.17%	6.1%	1455.7	0.22%	9.5%	1938.9	0.45%	13.5%	2482.0
IBM14	0.09%	5.8%	2082.7	0.29%	10.5%	3395.1	0.35%	15.0%	4322.3
IBM15	0.11%	4.5%	2305.6	0.37%	9.2%	3724.5	0.49%	12.3%	4506.1
IBM16	0.07%	6.7%	3385.1	0.15%	9.7%	3851.8	0.32%	15.8%	4815.9
IBM17	0.03%	6.5%	3932.0	0.14%	11.0%	4829.1	0.20%	16.7%	6137.5
IBM18	0.05%	5.7%	2970.5	0.12%	11.5%	4521.9	0.18%	15.2%	4880.7
Average	0.10%	5.9%	1072.2	0.24%	11.0%	1570.7	0.35%	15.5%	2108.2

CHAPTER V

UNIFIED ADAPTIVITY OPTIMIZATION OF CLOCK AND LOGIC SIGNALS

VLSI design is increasingly sensitive to variations which often degrade the parametric yield. Post-silicon tuning techniques can compensate for specific variations on the die and thus significantly improve the yield. Previous works on adaptivity optimization for post-silicon tuning focus on either logic signal tuning or clock signal tuning. This paper proposes the first unified adaptivity optimization on logical and clock signal tuning, which enables us to significantly save resource. In addition, it does not need any assumption on variation distributions.

Our unified optimization is based on a novel linear programming formulation which can be efficiently solved by an advanced robust linear programming technique. Due to the discrete nature of the problem, the continuous solution obtained from linear programming is then efficiently discretized. This procedure involves binary search accelerated dynamic programming, batch based optimization, and Latin Hypercube sampling based fast simulation. Our experimental results demonstrate that up to 50% area cost reduction can be obtained by the unified optimization compared to optimization on logic or clock alone. In addition, the proposed discretization approach significantly outperforms the alternatives in terms of solution quality and runtime.

A. Introduction

With continuously shrinking features on the die, VLSI design is increasingly sensitive to variations such as manufacturing process variations. Consequently, circuit performance is no longer determined solely by deterministic values. It has significant uncertainty which needs to be considered in order to achieve high yield. There exist a handful set of approaches (e.g., [54, 8, 55, 9, 10, 56]) which focus on performing sta-

tistical circuit optimization in the pre-silicon phase. That is, circuit parameters are determined in design time for yield optimization. With statistical variation models, they obtain the statistically optimized design and apply the design to all the dies. Although the optimized design is of good quality in statistical sense, the design is not necessarily ideal for each individual fabricated chip. Specific circuit parameter variations on the die cannot be mitigated. In addition, reliable statistical variation models are not easy to obtain [11].

In contrast to pre-silicon statistical optimizations, post-silicon tuning methodology can tune some circuit parameters after the chip is fabricated. This enables us to mitigate the specific circuit parameter variations on the individual chip to satisfy the design target. As a result, the timing yield can be significantly improved [12, 11].

Clearly, it is highly desirable to perform circuit adaptivity optimization for post-silicon tuning. Since making a circuit element post-silicon tunable necessarily introduces overhead, adaptivity optimization for post-silicon tuning aims to provide large tunability with small overhead. Previous works focus on either logic signal tuning (e.g., [12, 13, 11]) or clock signal tuning (e.g., [14, 15]). Note that some approaches (e.g., [13, 15]) also consider to perform gate sizing in design time, however, no joint tuning on logic and clock signal is performed in post-silicon phase. These approaches are effective, however, the resource utilization is not necessarily efficient since the interaction between logic circuit and clock network is not explored. Performing unified adaptivity optimization on clock and logic signals has the potential to significantly reduce overhead while still having large tunability for achieving yield target.

Common post-silicon tuning techniques include adaptive body biasing (ABB) [12] and tunable clock buffer (PST buffer) [14]. ABB is a well-established technique for tuning the body voltage of a circuit block to achieve different timing-power tradeoff [12]. Due to well-spacing related layout rules and overhead issue, it is desired to

apply ABB at circuit block level but not to individual device [11]. A PST buffer [14] can change its delay after fabrication by padding different amount of loads. In this paper, as an illustration of our methodology, ABB is used to tune logic signals as in [12, 13, 11] and PST buffer is used to tune clock signals as in [14, 15]. Our approach can be easily extended to handle other tuning techniques.

In this work, a unified adaptivity optimization technique on clock and logic signals is proposed. The new technique determines the location and the tuning range of each tunable circuit element. We propose methods for both continuous optimization and discrete optimization. Continuous approach assumes that each circuit element can be tuned to arbitrary precision. It involves a linear programming with uncertainty problem, which is solved by a robust linear programming approach with a parameter easily controlling the tradeoff between the worst-case design and the nominal design. Discrete approach discretizes the continuous solution, i.e., maps the tuning range of each tunable element to a permissible set of tuning ranges. It involves binary search accelerated dynamic programming, batch-based optimization, and Latin Hypercube sampling based fast simulations. Our main contribution is summarized as follows.

- According to the best of the authors' knowledge, this is the first work on unified adaptivity optimization for post-silicon tuning on clock and logic signals.
- In contrast to most previous works (e.g., [13, 11]), our approach computes the discrete solution in addition to the continuous solution. This is desirable due to the discrete nature of problem.
- Unlike many previous works (e.g., [13]), our new approach does not assume any variation distributions since reliable variation model is not easy to obtain in reality.

Computation cost is also an important issue. In our unified adaptivity optimiza-

tion approach, the problem size is almost the same as adaptivity optimization on clock signal alone. This is due to that logic tuning is applied to the circuit block level but not to individual gate (compared to e.g., [15]). We have up to several dozens of circuit blocks. In addition, our continuous approach does not perform any Monte Carlo simulation and thus runs very efficiently. Even for the discretization approach which involves Monte Carlo simulations, since the search space is largely reduced due to being guided by continuous solution, together with various acceleration techniques, it still runs efficiently. Although the works of [14, 15] are also independent of assumptions on variation distributions, they tend to be slow as full-fledge Monte Carlo simulations are frequently called during their optimization procedures.

Our experiments demonstrate that the new continuous unified adaptivity optimization approach is consistently better than adaptivity optimization on logic or clock signal alone. One can achieve up to 50% area cost reduction by our approach. Our discretization approach also significantly outperforms the alternatives including nearest rounding approach and binary batch approach [14] in terms of yield, area and runtime.

It is worth noting that the proposed methodology is flexible to use. Although it is mainly used for unified optimization on clock and logic signals, it can be easily reduced to optimizing only one of them. Since our continuous optimization method is fast, it can be easily integrated with other deterministic or statistical pre-silicon optimization, in an interactive manner, to achieve the joint pre and post silicon optimization (which is similar to [13, 15]). For simplicity, area overhead is considered in this paper. However, the proposed approach is not restricted to area overhead and can be extended to handle other types of overhead.

The rest of the chapter is organized as follows: Section B presents the motivation and the problem formulation of the work. Section C describes the overall flow of

the algorithm. Section D describes the continuous unified adaptivity optimization approach. Section E describes the discretization approach. Section F presents the experimental results with analysis. A summary of work is given in Section G.

B. Preliminaries and Motivation

In our adaptivity optimization approach, adaptive body biasing is applied to tune logic signals and PST buffer tuning is applied to tune clock signals. Note that as indicated in [11], due to well-spacing related layout rules and overhead issue, we apply ABB at circuit block level but not to individual device.

For adaptivity optimization on logic signals, body bias tuning is a well-established technique to tune the body voltage of transistors to achieve tradeoff between power and delay. Forward body biasing reduces threshold voltage and delay while increases power. In contrast, reverse body biasing increases threshold voltage and delay while reduces power. It is demonstrated in [11] that delay can be well approximated by a linear function of the body voltage. Due to well-spacing related layout rules and overhead issue, it is desired to apply ABB at circuit block level but not to individual device [11]. Refer to [11] for some clustering approaches to group gates into clusters for body biasing.

For adaptivity optimization on clock signals, PST buffers [14] are used. A PST buffer consists of two inverters with a set of load capacitors in between and the delay can be changed through controlling the gates on capacitors. It is able to produce uniform delay step in clock tuning and provide good tuning range. Refer to [14] for the details.

Since making a circuit element (e.g., a gate or a clock buffer) post-silicon tunable necessarily introduces overhead, adaptivity optimization for post-silicon tuning aims

to provide large tunability with small overhead. Previous works focus on logic signal (e.g., [12, 13, 11]) or clock signal (e.g., [14, 15]) separately. Some approaches (e.g., [13, 15]) also consider to perform gate sizing in design time, however, no joint tuning on logic and clock signal is performed in post-silicon phase.

These approaches are effective, however, their resource utilization is not necessarily efficient since the interaction between logic circuit and clock network is not explored. Consider that we have a circuit which can achieve 99% yield by logic signal tuning alone. The observation is that some slack can be moved from non-timing critical part to timing critical part by introducing useful skew using clock signal tuning, and clock signal tuning is often cheaper than ABB (due to e.g., ABB is applied on a block of gates, a lot of more control signals need to be used, and ABB needs Digital-to-Analog converters [12]). As a result, usage of clock signal tuning would lead to less logic tuning and thus significantly reduce the overhead. On the other hand, solely relying on clock signal tuning often leads to small improvement on yield. This is the case since there may be many critical cycles in a circuit, and when these cycles are heavily overlapped, one can only move small amount of slack among flip-flops. In addition, a single clock buffer may drive many flip-flops and tuning it affects all of them. If some flip-flops are in timing critical paths, then only small amount of tuning can be applied to this clock buffer.

The motivation of the unified optimization is further illustrated by the example of Figure 33. Figure 33 shows a sequential circuit consisting of 4 flip-flops $FF1, FF2, FF3, FF4$, two clock buffers, and five combinational paths A, B, C, D, E. The nominal delay for each combinational path is shown and all clock skews are zero. Suppose that the clock period is 10 and each combinational path has 10% variation off the nominal delay value. For simplicity, assume that we need to guarantee the worst-case design satisfying the timing constraint. Since three combinational paths

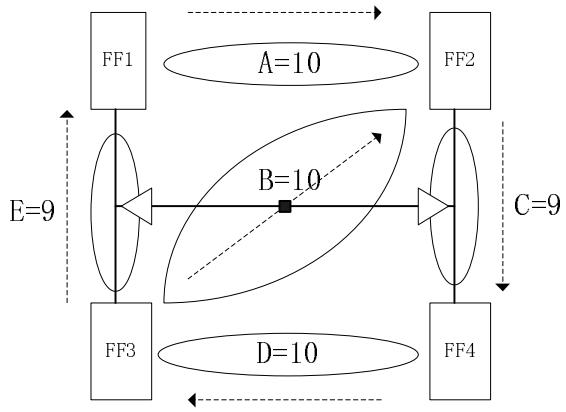


Fig. 33. A sequential circuit where the arrows show the signal flow directions. The central square is the clock source and the triangles are clock buffers.

A, B, D may have timing violations, we need to tune them by ABB. If clock tuning is also used, we can tune the right clock buffer to introduce 1 unit delay, i.e., $FF2$ and $FF4$ both have skew of 1. One can see that A, B do not need ABB due to the additional slack. That is, tuning the right clock buffer and D is sufficient, which certainly saves overhead. It is also clear that tuning clock signal alone may not make both A and D satisfy the timing constraint.

As shown above, performing unified adaptivity optimization on clock and logic signals has the potential to significantly reduce overhead while still having large tunability for achieving yield target. From our experience with ISCAS'89 benchmark circuits, by unified adaptivity optimization, up to 50% area cost reduction can be obtained compared to adaptivity optimization on logic or clock signals alone.

For convenience, a gate with tunable body voltage (in logic circuit) or a buffer with programmable delay (in clock tree) is called a tunable element. All tunable gates in a circuit block have the same amount of body voltage tuning. Circuit blocks can be formed by clustering gates as in [11] or in any other way. Our approach is independent of circuit partitioning method. Our problem is formulated as follows.

Unified Adaptivity Optimization Problem: Given a sequential circuit, we are to perform adaptivity optimization on logic and clock signals, i.e., decide the location and the tuning range of each tunable element, such that the area cost overhead is minimized subject to the constraint that the fabricated circuits can be tuned to satisfy the yield target by post-silicon tuning.

$$\min \quad c_{b,12} + n_{fbb,12} - n_{rbb,12} \tag{5.1}$$

$$\begin{aligned} s.t. \quad & (S_1 + d_1) + (T_{12} - n_{12}D_{p_{12}} - n'_{12}D_{p_{12}}) - (S_2 + d_2) \\ & \leq T_{cp} - T_{setup}, \end{aligned} \tag{5.2}$$

$$\begin{aligned} & (S_1 + d_1) + (t_{12} - n_{12}D_{p_{12}} - n'_{12}D_{p_{12}}) - (S_2 + d_2) \\ & \geq T_{hold}, \end{aligned} \tag{5.3}$$

$$c_{b,12} = m_{b_1}C_{b_1} + c_{b_1} + m_{b_2}C_{b_2} + c_{b_2} + m_{b_3}C_{b_3} + c_{b_3}, \tag{5.4}$$

$$n_{fbb,12} = n_{12}C_{p_{12}} + c_{p_{12}}, \tag{5.5}$$

$$n_{rbb,12} = n_{12'}C_{p_{12}} + c_{p_{12}}, \tag{5.6}$$

$$d_1 = m_{b_3}D_{b_3} + m_{b_1}D_{b_1}, \tag{5.7}$$

$$d_2 = m_{b_3}D_{b_3} + m_{b_2}D_{b_2}, \tag{5.8}$$

$$0 \leq n_{12} \leq U_{n_{12}}, \tag{5.9}$$

$$-U_{n_{12}} \leq n'_{12} \leq 0, \tag{5.10}$$

$$0 \leq m_{b_1}, m_{b_2}, m_{b_3} \leq U_m, \tag{5.11}$$

$$S_1, S_2, T_{12}, t_{12} \text{ are random variables.} \tag{5.12}$$

C. Overall Flow

A two-stage optimization approach is proposed to decide the location and the tuning range of each tunable element in the unified adaptivity optimization. The first stage is called *Continuous Optimization* which is to efficiently compute a post-silicon tunable design with the assumption that one can achieve arbitrary tuning precision for any tunable element. The second stage is called *Discretization*, which is to map the obtained continuous tuning range for each tunable element into the permissible (i.e., discrete as in reality) set of tuning ranges.

D. Continuous Optimization

The input to the first stage is the statistical timing analysis results on the given sequential circuit. To decide the locations and the tuning ranges of ABB tunable blocks and PST clock buffers, a linear programming with uncertainty problem is formulated and solved using a robust linear programming technique with guaranteed constraint violation probability bound. Linear programming formulation is used since delay can be well approximated by a linear function of overhead [11].

1. Linear Programming Formulation

A sequential circuit is represented as a timing graph where each node represents a flip-flop and a directed edge from node u to node v represents the combinational logic paths from u to v. We only describe the mathematical formulation for a single edge (together with two nodes) in a timing graph. Other edges can be similarly handled. Refer to Figure 34 for part of a given clocked circuit. Suppose that in Figure 34, Suppose that in a clocked circuit, two flip-flops FF_1 and FF_2 are connected by combinational paths. The clock delay at FF_1 is S_1 and at FF_2 is S_2. The long

(resp. short) critical combinational path delay between FF_1 and FF_2 is T_{12} (resp. t_{12}). All of S_1, S_2, T_{12}, t_{12} are random variables.

Without loss of generality, we assume that the combinational path p_{12} connecting FF_1 and FF_2 passes one circuit block. [11] demonstrates that in body biasing, delay reduction is linear with body voltage tuning. This fact is used here. To tune the path by ABB, we assume that the delay reduction due to body biasing along p_{12} is $n_{12}D_{p_{12}}$[1]. By linear fitting to the delay-area overhead data, $D_{p_{12}}$ is the slope of fitted line. For convenience, we call it "unit" delay reduction. n_{12} denotes the amount of ABB unit delay reduction applied to the circuit. The delay reduction comes with the area overhead due to e.g., control logic, extra well space, and extra power wires. It is measure by $n_{12}C_{p_{12}} + c_{p_{12}}$, where $C_{p_{12}}, c_{p_{12}}$ are also obtained from linear fittings. We call $C_{p_{12}}$ unit area overhead, and constant $c_{p_{12}}$ comes from e.g., shared control logics for the block of tunable elements. Similarly, we assume that the delay increase due to post-silicon tuning of PST clock buffer b_i is $m_{b_i}D_{b_i}$ with the area overhead $m_{b_i}C_{b_i} + c_{b_i}$ where D_{b_i} (resp. C_{b_i}) is the unit delay reduction (resp. the unit area overhead) for tuning a PST clock buffer, and c_{b_i} is constant indicating the overhead of control signals. The unified adaptivity optimization problem can be formulated as in Eqn. (5.1)-Eqn. (5.12).

Note that d_1 in Eqn. (5.7) and d_2 in Eqn. (5.8) are clock delay tuning at FF_1 and FF_2, respectively, and $n_{fbb,12}$ and $n_{rbb,12}$ are area overhead due to forward body biasing and reverse body biasing, respectively. $c_{b,12}$ is the area overhead due to clock tuning. They are introduced for the clarity of the formulation. S_1, S_2, T_{12}, t_{12} are the random variables obtained from statistical timing analysis. Only $m_{b_1}, m_{b_2}, m_{b_3}, n_{12}, n_{12'}$ are the decision variables in the formulation. Loosely speaking, $m_{b_1}, m_{b_2}, m_{b_3}$ are the

[1]Note that if reverse body biasing is used, the delay is increased, i.e., delay reduction becomes negative.

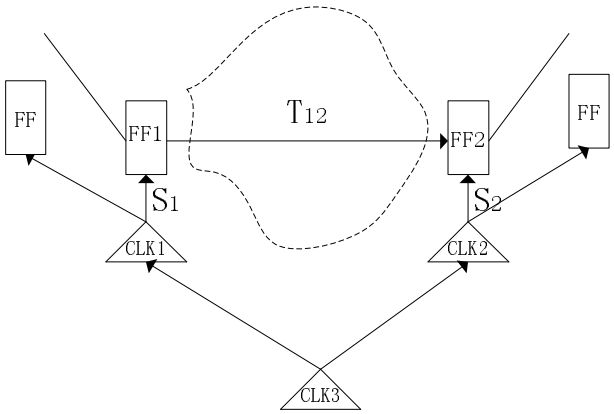


Fig. 34. Part of a sequential circuit. The dotted region is the circuit block.

amount of unit tuning applied on each clock buffer, n_{12} is the amount of unit forward body bias tuning applied on each circuit block (noting that they are non-negative), and n'_{12} is the amount of unit reverse body bias tuning applied on each circuit block (noting that they are non-positive). They can be any real values within certain ranges when solving the above linear programming problem. T_{cp} is the clock period, and T_{setup} and T_{hold} are the setup hold and the hold time of FF_2, respectively. Constants $U_{n_{12}}, U_m$ are the maximum tuning ranges which are allowed for each tunable element.

We make the following comments on the problem formulation. First, the meaning of variables like n_{12} is the *tuning* for each circuit. For example, for a fabricated circuit with some deterministic values of S_1, S_2, T_{12}, t_{12}, variables like n_{12} are the desirable tuning for the circuit by which the circuit can be tuned to satisfy the timing constraint. Since S_1, S_2, T_{12}, t_{12} are random variables, after solving the linear programming with uncertainty problem, the meanings of the obtained $n_{12}, n'_{12}, m_{b_1}, m_{b_2}, m_{b_3}$ become the *tuning ranges* since they need to guarantee that most (determined by yield target) fabricated circuits can be tuned to satisfy the timing constraint. The actual tuning on each fabricated circuit can be smaller than the tuning ranges.

Second, the first two constraints (i.e., Eqn. (5.2) and Eqn. (5.3)) specify the long

path constraint and the short path constraint, respectively. For a path with tight long path constraint and loose short path constraint, n_{12} will be non-zero and n'_{12} will be zero. For a path with tight short path constraint and loose long path constraint, n'_{12} will be non-zero and n_{12} will be zero. For a path with tight long path constraint and tight short path constraint, all of four variables may be non-zero.

Eqn. (5.2) and Eqn. (5.3) contain random variables. To solve the linear programming with uncertainty, they will be cast into deterministic constraints using a technique proposed in the robust linear programming literature [57]. With theoretical guarantee, this technique enables us to specify a tradeoff parameter controlling the probability of constraint violation which is strongly related to timing yield. Such an approach has also been successfully used in [58] for clock scheduling. In our approach, we sample the subcircuits for simulations and use the results to decide the tradeoff parameter. That is, our usage of robust linear programming is adaptive.

2. Robust Linear Programming

Consider a general linear programming problem [57]

$$\min \quad \mathbf{c}'\mathbf{x} \tag{5.13}$$

$$s.t. \quad \sum_{j=1}^{n} \mathbf{A}\mathbf{x} \leq b, \tag{5.14}$$

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \tag{5.15}$$

where $\mathbf{A}$ contains uncertainty, i.e., some coefficients in the constraints could be random variables. In [59], Soyster proposes a method which solves it assuming that all random variables can simultaneously take the worst-case values. Let J_i denote the set of coefficients in row i of A which contain uncertainty. Under the assumption that each entry $a_{ij} \in J_i$ could take any value from $[\mu_{a_{i,j}} - \hat{a}_{i,j}, \mu_{a_{i,j}} + \hat{a}_{i,j}]$, Soyster's method

guarantees that the computed solution will always satisfy all the constraints. $\mu_{a_{i,j}}$ is the nominal value and if a coefficient does not have any uncertainty, $\mu_{a_{ij}} = a_{ij}$. $\hat{a}_{i,j}$ ($\hat{a}_{i,j} \geq 0$) is closely related to the standard deviation σ of the probability distribution. For example, $\hat{a}_{i,j}$ could be set to 3σ or 6σ depending on how strict the yield requirement is.

Soyster's method is essentially the worst-case design methodology which could introduce large resource overhead unnecessarily. A desirable approach should be able to achieve different tradeoff between the nominal-case design and the worst-case design. Bertsimas and Sim [57] propose such a technique, which enables us to cast the linear constraints with uncertainty into deterministic linear constraints. In addition, they have a **tradeoff parameter**, denoted by p, which controls the degree of conservatism so as to obtain different tradeoff. Such an approach has also been successfully used in [58] for clock scheduling. The above linear programming problem is first cast into [57]

$$\min \quad \mathbf{c}'\mathbf{x} \tag{5.16}$$

$$s.t. \quad \sum_j \mu_{a_{ij}} x_j + \max_{S_i \cup \{t_i\} | S_i \subseteq J_i, |S_i| = \lfloor p \rfloor, t_i \in J_i / S_i} \{ \sum_{j \in S_i} \hat{a}_{ij} y_j + (p - \lfloor p \rfloor) \hat{a}_{it_i} y_t \} \leq b_i, \forall i \tag{5.17}$$

$$-y_j \leq x_j \leq y_j, \tag{5.18}$$

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \tag{5.19}$$

$$\mathbf{y} \geq 0. \tag{5.20}$$

Informally, the new constraint in the above formulation means that one could have at most p ($p \leq |J_i|$) variables simultaneously achieving the worst-case values and the deviations of the other variables off their nominal values are controlled by $p - \lfloor p \rfloor$. In our problem, since every constraint with uncertainty (e.g., Eqn. (5.2) and Eqn. (5.3))

contains the same number (i.e., 3) of random variables, p will be a real value between $[0, 3]$. The above nonlinear programming problem is shown to be equivalent to [57]

$$\min \quad \mathbf{c}'\mathbf{x} \tag{5.21}$$

$$s.t. \quad \sum_j \mu_{a_{ij}} x_j + z_i p + \sum_{j \in J_i} q_{ij} \leq b_i, \forall i \tag{5.22}$$

$$z_i + q_{ij} \geq \hat{a}_{ij} y_j, \forall i, j \in J_i \tag{5.23}$$

$$-y_j \leq x_j \leq y_j, \tag{5.24}$$

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \tag{5.25}$$

$$\mathbf{q}, \mathbf{y}, \mathbf{z} \geq 0. \tag{5.26}$$

Clearly, when $p = 0$, z_i can be $\hat{a}_{ij} y_j$ so that $q_{ij} = 0$ and thus $z_i p + \sum_{j \in J_i} q_{ij} = 0$. That is, the problem becomes the nominal-case design. When $p = 3$, every z_i counts and $z_i p + \sum_{j \in J_i} q_{ij}$ is simply $\sum_{j \in J_i} \hat{a}_{ij} y_j$. Thus, the problem becomes the worst-case design. Varying p in between, different tradeoff can be obtained. Let $\mathbf{x}^*$ be the optimal solution obtained by the above method. [57] proves that the constraint violation probability is exponentially decreased with p. Precisely,

$$\mathtt{Pr}(\sum_j a_{ij} \mathbf{x}^* > b_i) < e^{-\frac{p^2}{6}}, \forall i. \tag{5.27}$$

[57] also demonstrates this through extensive experiments.

3. Adaptive Application of Robust Linear Programming

In our case, since the variations are not assumed to follow any specific distribution, tradeoff parameter p cannot be analytically determined for satisfying the yield target. For large circuits, one could perform simulations according to different p on the whole circuit to find the best tradeoff. However, this is time consuming. We propose to use the following circuit sampling based adaptive approach for determining p. For

large circuits in the experiment, some sub-circuits are first sampled and the above formulation is performed on them. Subsequently, for different p, we solve the linear programming problem and perform Monte Carlo simulations to obtain the constraint violation probability. p is then experimentally determined so as to provide the best degree of conservatism. Precisely, we choose the smallest p such that the timing yield is equal to or larger than the yield target. Since sampled circuits have similar characteristics as the whole circuit, we wish that using this p we would be able to compute the solution with similar yield for the whole circuit. In this sense, our usage of robust linear programming is adaptive.

E. Discretization

In reality, it is very rare that a tunable delay element, either ABB or PST buffer, can be tuned continuously. In other words, the tuning is often allowed only for certain discrete steps. The second stage of the algorithm is to map the solution of the linear programming problem into a permissible (discrete) solution. Rounding up the tuning range of every tunable element to the continuous solution will obtain a discrete solution which satisfies the yield target but with more area cost overhead. In contrast, rounding down these tuning ranges could obtain a discrete solution with smaller area but not satisfying the yield target. If the size of each tuning step is small, nearest rounding may provide acceptable discrete solutions. Otherwise, with considerable size of the tuning step, nearest rounding may result in large error and a dedicated discretization algorithm is highly desirable. In our discretization process, we are to decide which one of the two choices should be used at each tunable element. This process consists of PST clock buffer tuning range rounding (i.e., clock rounding) and logic circuit tuning range rounding (i.e., logic rounding). Clock rounding is first

performed and returns a set of solutions where all clock tuning ranges are discretized while logic tuning ranges are still continuous. Logic rounding is then performed to some of the above solutions to discretize their logic tuning ranges.

1. Discretizating PST Clock Buffers

A dynamic programming approach is first used to determine discrete tuning range for each PST buffer in the clock tree. We define a *partial clock tuning solution* to be an incomplete determination for the discrete tuning ranges of all clock buffers. A partial clock tuning solution becomes *clock-complete* when the discrete tuning ranges of all clock buffers are determined. A clock buffer is *processed* if its discrete tuning range has been determined. The algorithm starts with the root of the clock tree, and performs a breadth-first traversal on the clock tree. During the process, we set the tuning range for each tunable clock buffer to each of two possible choices (up-rounding and down-rounding). The acceleration technique is necessary.

During the process, we set the tuning range for the first tunable clock buffer to each of two possible choices (up-rounding and down-rounding), which results in two partial solutions. For each partial solution, we process the second tunable clock buffer and set its tuning range to each of two choices. In this way, the algorithms proceeds in a dynamic programming fashion. That is, it processes each tunable clock buffer in turn according to the breadth-first order. Without any acceleration technique, there are 2^m solutions for optimizing a clock tree with m PST buffers. Therefore, during the solution propagation process, inferior solutions are pruned for acceleration. The algorithm terminates when all partial solutions become clock-complete.

a. Solution Characterization

A set of partial solutions, denoted by $\mathcal{A}$, keep being updated during the process of dynamic programming. Each solution $\alpha \in \mathcal{A}$ is associated with a (C, Y) pair, where C denotes the cumulative area overhead and Y denotes the estimated yield. C is computed by summing the overhead of all processed PST buffers. To compute the yield of the circuit, discrete tuning range for every tunable element needs to be known. Since some of them are not processed, we will use their continuous tuning ranges for yield estimation. This makes sense as our goal for performing discretization is to obtain a discrete solution with yield and overhead close to the continuous solution.

b. Solution Propagation

Suppose that we are to decide the tuning range of a PST buffer b. A new solution α' will be formed for each of the two possible choices (up-rounding and down-rounding). Because all PST buffers are processed according to the breadth-first order, when b is processed, the cumulative area overhead can be updated by $C(\alpha') = C(\alpha) + C(b)$, where $C(b)$ is the area overhead due to b. $Y(\alpha')$ is obtained by simulations, precisely, a fast yield estimation through Latin Hypercube sampling based Monte Carlo simulations [60]. Refer to Section 3 for the details.

c. Acceleration by Pruning

The acceleration comes from the observation that we do not need to always update the yield of every partial solution during solution propagation.

After processing a node u, we obtain a set of partial solutions $\mathcal{A}_u$, each of which satisfies the yield target. Suppose that the next node to be processed is v. For each solution in $\mathcal{A}_u$, a new solution is generated by rounding up the PST buffer tuning

range at v. Denote the resulting solution set by $\mathcal{A}_{v,+}$. Late yield update is applied on them. That is, their yields are not computed at this moment since we know that they must satisfy the yield target. For each solution in $\mathcal{A}_u$, a new solution is also generated by rounding down the PST buffer tuning range at v. Denote the resulting solution set by $\mathcal{A}_{v,-}$, which is sorted by C values. We are to perform yield estimation on $\mathcal{A}_{v,-}$ since some of them may not meet the yield target. Our yield estimation is performed in a binary search fashion. That is, we first estimate the yield for the middle solution. If it satisfies the yield target, the middle solution for the half solution set with smaller C will be tested. Otherwise, the current solution and the half solution set with smaller C will be pruned, and the middle solution for the half solution set with larger C will be tested. The process is repeated for $\log |\mathcal{A}_{v,-}|$ times. Denote the resulting solution set by $\mathcal{A}'_{v,-}$ and then $\mathcal{A}_v = \mathcal{A}_{v,+} \bigcup \mathcal{A}'_{v,-}$. During solution propagation, when the size of the solution set $\mathcal{A}$ is larger than a threshold w, top $w/2$ solutions with smallest C values are kept and all other solutions are pruned for further speedup. By these techniques, our clock rounding approach can be significantly accelerated. After all the leaves in the clock tree are processed, a set of clock-complete solutions $\mathcal{A}$ are obtained. Which discrete solution is eventually selected depends on the logic tuning range discretization discussed in the next subsection.

2. Discretizing Logic Circuits

Section 1 returns a set of solutions, denoted by $\mathcal{A}$, which are sorted according to C values. For each solution in $\mathcal{A}$, clock tuning ranges have already been discretized while logic tuning ranges are not. This subsection deals with discretizing the logic tuning ranges and selecting which rounding solution to return.

The solution set $\mathcal{A}$ can be large and it is time consuming to perform logic rounding on each solution. Thus, similar to Section c, a binary search fashion algorithm

is applied on the solution set $\mathcal{A}$. That is, the middle solution is first rounded (by logic rounding) followed by the middle solution in one half of $\mathcal{A}$. Finally, the solution (where both clock rounding and logic rounding have been performed) to be returned is the one with the smallest overhead while satisfying the yield target. We only describe how to round a single solution.

Recall that our body bias tuning is applied at the circuit block level (i.e., the whole circuit block has the same tuning range), we will discretize the tunable ranges for tunable circuit blocks. Each tunable circuit block is assigned with a timing criticality related reducibility which measures the possible area overhead reduction while still satisfying yield target. A large reducibility means the large possibility of area reduction. Since larger slack means that we have larger room for area reduction, and the area overhead is also proportional to the number of tunable gates, the following cost function is used for a path $\mathfrak{p}$ passing through a block B.

$$reducibility(\mathfrak{p}) = slack(\mathfrak{p}) \times tunablegates(\mathfrak{p}), \tag{5.28}$$

where $slack(\cdot)$ is the sum of the slack of the gates along $\mathfrak{p}$ in B and $tunablegates(\cdot)$ is the number of tunable gates along $\mathfrak{p}$ in B. Note that the nominal slack is used as an estimation for the slack, which makes sense since variations in the block should have similar impact on all paths passing through it. Since all critical combinational paths are given as the input to our algorithm, the paths passing through a block can be easily identified. The reducibility of a block is then defined as the minimum reducibility cost for all paths passing through it since we need to guarantee that (almost) all paths could satisfy the timing constraint. That is,

$$reducibility(B) = \min_{\mathfrak{p}} reducibility(\mathfrak{p}). \tag{5.29}$$

To perform logic rounding, a batch-based optimization technique is used after

the reducibility costs for all blocks are computed. This technique is also used in [14] for clock tuning which is shown to be much more efficient than greedy approach. We set a threshold parameter k. For all circuit blocks with reducibility greater than k, the tuning ranges for the blocks are rounded down (from the continuous solution) and the tuning ranges for all other blocks are rounded up. Fast Latin Hypercube sampling based Monte Carlo simulations (refer to Section 3) are then performed to obtain the timing yield. We start from a small k. If the yield target is satisfied, the rounding is accepted. Otherwise, k is increased. The above process is repeated until the yield target is satisfied. Note that if each time k is set to the largest reducibility among all blocks, the batch-based optimization becomes the greedy approach. In this case, we may have to perform many iterations of the algorithm which is time consuming, and the approach can be easily stuck into local optimum. If k is set to the smallest reducibility among all blocks, it becomes the down-rounding approach, which cannot achieve the yield target. Varying initial k, we could achieve different tradeoff between area overhead and computation overhead.

3. Fast Simulations for Timing Yield Estimation

Discretization involves simulations for yield estimation. In simulations, the tuning range of a tunable element needs to be fixed. This is implemented as modifying the corresponding constraints in the linear programming formulation. For example, discretizing n_{12} in Eqn. (5.2) to 10 is implemented as setting $n_{12} \leq 10$ in Eqn. (5.9). During discretization, we may frequently estimate the new timing yield of the circuit (for not-yet discretized tuning ranges, their continuous tuning ranges are used). The commonly-used Monte Carlo simulation approach is very time consuming. Thus, fast Latin Hypercube (LH) sampling based Monte Carlo simulations are used. Compared to simple sampling, LH sampling allows us to sample the space more evenly to avoid

crowded samples. By this, one can use much fewer samples in Monte Carlo simulations while still having high yield estimation accuracy [60].

The common approach for yield estimation is to formulate a set of, say 5000, linear programming instances with uncertain variables set to some deterministic values according to probability distributions, and solve them to compute the yield as $Yield = \frac{\text{the number of feasible LP instances}}{5000}$. This full-fledged Monte Carlo simulation approach is very time consuming. Thus, fast Latin Hypercube (LH) sampling based Monte Carlo simulations are used. Compared to simple sampling, LH sampling allows us to sample the space more evenly to avoid crowded samples. By this, one can use much fewer, typically 200, samples in Monte Carlo simulations while still having high yield estimation accuracy [60].

For simplicity, we use a two-dimensional simulation example to illustrate the idea of LH sampling. Given two random variables x, y for performing simulations, to generate 5 simulation instances is to randomly generate 5 pairs of x, y. As one can see from Figure 35(a), these instances could be not evenly distributed in the simulation space and not span the whole space. This is due to that each instance does not have any memory of previously generated instances. Crowded instances should be avoided for improving the accuracy and reducing the number of simulations.

LH sampling can effectively tackle this issue. In LH sampling, we divide the range of each variable into equal-probability intervals and requires that there is exactly one instance covering each interval in simulations. For example, in Figure 35(b), we first divide the simulation space into 5×5 equal-probability grids and whenever one grid is picked for simulation, the corresponding row and column will be removed from future simulation instance generation. In this way, the instances could be distributed more evenly than purely random instance generation approach, and thus the total number of simulations can be significantly reduced. Refer to [60] for the details and to [61]

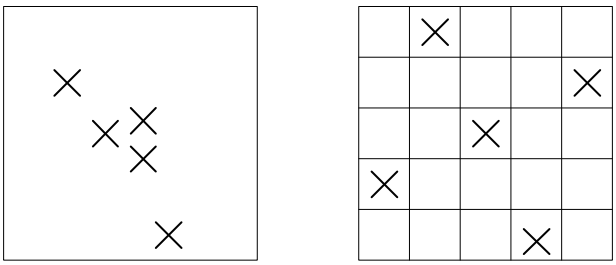


Fig. 35. Left: random sampling. Right: LH sampling. $\times$ denotes a sample.

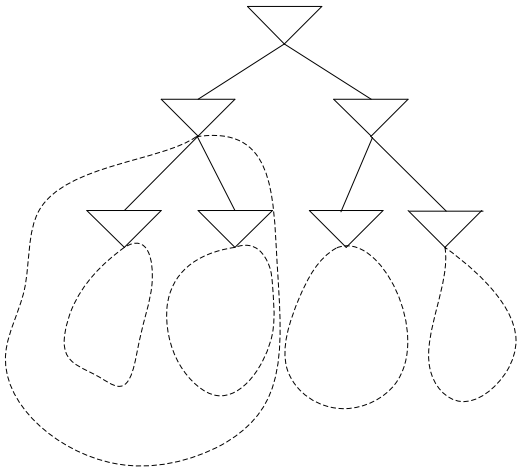


Fig. 36. A three-level clock tree. Dotted region refers to the covered circuit by the clock buffer.

for a successful application in CAD.

4. Time Complexity

In our algorithm, the main portion of runtime is certainly due to simulations in the discretization stage. Let r denote the runtime for one yield estimation using a set of 200 LH sampling based Monte Carlo simulations on the whole circuit. We will express the time complexity of the algorithm by r. The first time-consuming part is clock buffer discretization. Suppose that an H-tree is used as clock tree and there are totally l levels in the clock tree (refer to Figure 36 for a three-level clock tree). At

each node of the clock tree, we only need to perform the simulations on the circuits covered by all the leaves (flip-flops) of its subtree, since changing the tuning range of the node will not affect the part not covered by its leaves. Thus, one set of simulations on the whole circuit is needed for each level of the clock tree, which takes r time. For an l-level clock tree, the runtime is lr for a single solution. Assume that there are on average $|\mathcal{A}_{avg}|$ solutions during the solution propagation. The total runtime is $\log|\mathcal{A}_{avg}|lr$.

The second time-consuming part is circuit block discretization. Assuming that totally q iterations of batch-based optimization is performed. In each batch, all paths passing through the selected circuit blocks need to be simulated. The total runtime is bounded above by qr since at most one set of simulations on the whole circuit is needed for a batch. Thus, the runtime is $\log|\mathcal{A}_{avg}|qr$ assuming that $|\mathcal{A}_{avg}|$ solutions are obtained from clock tree discretization. In a total, the runtime of the algorithm is bounded above by $\log|\mathcal{A}_{avg}|(l+q)r$. For large circuits, $l+q$ is around 15 and $\log|\mathcal{A}_{avg}|$ is quite small (< 8) through controlling w in Section c which gives very good tradeoff between solution quality and runtime.

F. Experiments

The continuous linear programming algorithm and discretization algorithm are implemented in C++ and are tested on a Pentium IV computer with a 3.0GHz CPU and 2G memory. ISCAS'89 benchmark circuits and a cell library of $130nm$ technology are used in the experiments. Logical synthesis and technology mapping (using Berkeley SIS) are performed to the circuits with a cell library of $130nm$ technology. The circuits are placed using Cadence Silicon Ensemble. H-tree topology is used to generate clock tree for the placement. Note that our approach can be easily applied

to other buffered clock tree topology. In the experiments, delay variations on gates and clock buffers are assumed to follow normal distributions with the standard deviation set to 5% of the mean value. Note that our approaches are independent of variation distributions. Statistical timing analysis is then performed whose runtime is not included in the algorithms since it is assumed to be an input of our algorithms.

In our experiments, the yield target is set to 99.0%. In our continuous approach, yield estimation is implicitly used in the robust linear programming technique. In our discretizartion approach, Latin Hypercube sampling based Monte Carlo simulations are used for yield estimation. These yield estimations are very fast, but may have small errors. In order to compensate for such errors, we set the yield constraint in the optimization to be 99.5% which is slightly higher than our target of 99.0%. This *target compensation* idea can compensate for estimation errors and make our results satisfy the yield target in practice. After optimization, 5000 Monte Carlo simulations are performed to evaluate the timing yield for the obtained circuits. Their runtime is not included in the algorithms since they are not in optimization.

1. Continuous Adaptivity Optimization

Our first experiment is to investigate the difference between the unified optimization and the optimization on logic or clock signal adaptivity alone. Recall that our continuous optimization problem is solved using a robust linear programming technique. A single *tradeoff parameter* is used to achieve different cost-yield tradeoff and no Monte Carlo simulation is needed during optimization. The optimization for logic or clock adaptivity alone follows the above procedure except that additional constraints are introduced to ensure that the optimization is not performed for both clock and logic signals simultaneously. The information of ISCAS'89 benchmark circuits is in Table XIX and the optimization results are summarized in Table XX. Note that due to

the target compensation idea, all results satisfy the yield target 99.0%. We make the following observations:

- Unified optimization saves large amount of area compared to optimization on logic signal or clock signal alone. For s5378, 50.5% area cost reduction is obtained.
- Clock Signal Adaptivity leads to small improvement on yield. Thus, linear programming often returns no feasible solutions for the same tradeoff parameter as in the unified optimization and Logic Signal Adaptivity. This is the case since there may be many critical cycles in a circuit, and when these cycles are heavily overlapped, there is only slight of amount of slack can be moved among flip-flops. In addition, a single clock buffer may drive many flip-flops and tuning it affects all of them. If some flip-flops are in timing critical paths, then only small amount of tuning can be applied to the clock buffer.
- Since the runtime only comes from formulating the linear program and solving it using the robust linear programming technique, all algorithms run very efficiently.

With different tradeoff parameter, we are able to obtain different tradeoff between cost (i.e., area overhead) and yield. Figure 37 shows a cost-yield curve obtained from tuning tradeoff parameter in our unified optimization approach. This curve provides large freedom for satisfying various yield and overhead requirement.

2. Discretization

We then perform discretization algorithm to the continuous solution obtained from the unified optimization. Since there is no previous work on unified optimization on

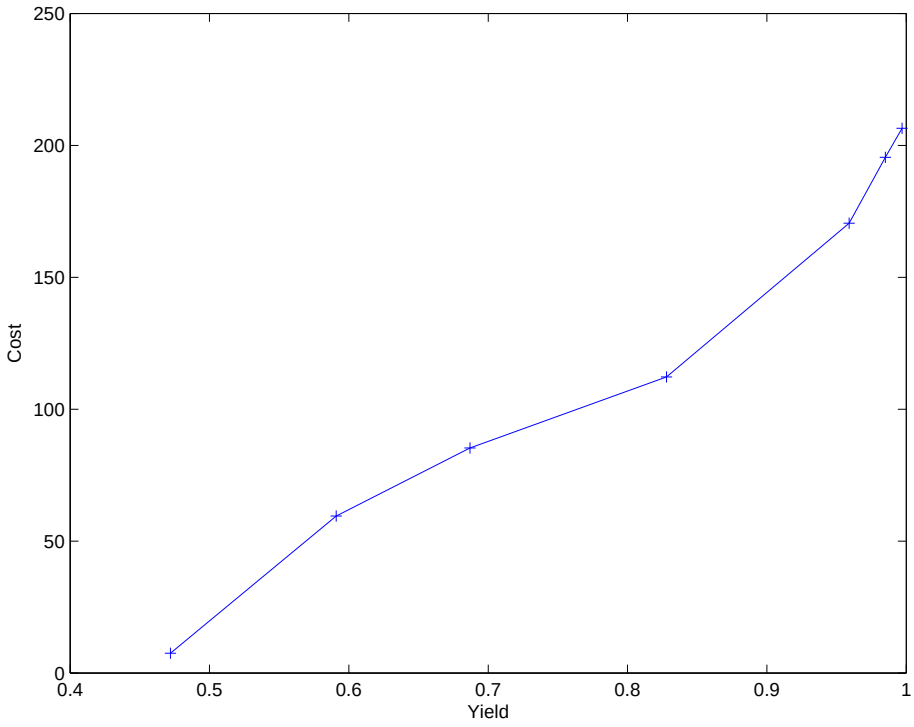


Fig. 37. Cost-Yield tradeoff curve for s1423 by the unified optimization.

Table XIX. Statistics of ISCAS'89 benchmark circuits. #Bk refers to the number of blocks and #Buf refers to the number of clock buffers.

Name	#FF	#Gates	#Bk	#Buf
s838	32	446	25	45
s1238	18	508	25	23
s1423	74	657	25	72
s1488	6	653	25	16
s5378	179	2779	25	128
s15850	534	9835	49	239
s35932	1728	16065	64	438
s38417	1636	22179	64	393
s38584	1426	19279	64	357

the adaptivity of logic and clock signals, we compare our approach to the following two simple methods. The first approach is called Binary Batch algorithm which is in the same spirit as [14]. Initially, we plan to compare to a greedy algorithm where each circuit block is tuned separately. However, we found that this approach is intolerably slow. The batch based optimization [14] gradually increases the tuning range of a set of blocks where the blocks are picked according to the criticality. In order to further improve its efficiency, binary search is performed on the tuning steps (i.e., tuning range can be increased by various multiples of discrete tuning steps) to largely reduce the total number of Monte Carlo simulations. Initially, tuning ranges of all blocks are 0 and thus the continuous solution is not used here. The tuning ranges of a set of blocks are then increased by a discrete level which is equal to a few multiples of the step size. It the yield target is not satisfied, we keep increasing the tuning ranges. Otherwise, we rip-up the solution and try a smaller discrete level on fewer blocks until the discrete level reaches the step size. The second algorithm for comparison is a Nearest Rounding approach. In this approach, the tuning range of each tunable element is simply rounded to the nearest discrete tuning range. All approaches involve

Table XX. Continuous optimizations on ISCAS'89 benchmark circuits. #Bk refers to the number of blocks and #Buf refers to the number of clock buffers. Area reduction is obtained by comparing the area of Our Discrete Solution with the minimum area of Logic Signal Adaptivity and Clock Signal Adaptivity.

Circuit	Logic Signal Adaptivity			Clock Signal Adaptivity			Unified Adaptivity			
Name	Area	Yield	CPU (s)	Area	Yield	CPU (s)	Area	Yield	CPU (s)	Area red.
s838	185.1	99.7%	5.5	-	-	-	103.7	99.8%	5.5	43.9%
s1238	204.1	99.8%	1.1	218.5	99.8%	1.1	112.2	99.9%	1.1	45.0%
s1423	216.2	99.4%	51.2	-	-	-	206.5	99.7%	51.5	4.5%
s1488	230.8	99.1%	1.2	-	-	-	201.9	99.3%	1.2	12.5%
s5378	207.5	99.3%	60.9	213.7	99.2%	61.2	102.7	99.5%	61.1	50.5%
s15850	1081.7	99.4%	342.5	-	-	-	703.6	99.3%	341.7	34.9%
s35932	2077.5	100.0%	1007.8	-	-	-	1532.8	99.7%	1012.5	26.2%
s38417	1728.2	99.3%	570.5	-	-	-	1339.7	99.5%	569.4	22.5%
s38584	2593.2	99.7%	667.5	-	-	-	1820.8	99.5%	668.5	29.8%

Latin Hypercube sampling based Monte Carlo simulations [60] and 200 LH samples are used in simulations for a yield estimation. Note that the yield constraint is pushed to 99.5% by our target compensation idea.

Table XXI. Discrete Solutions for ISCAS'89 benchmark circuits with large tuning step. Runtime for computing nearest rounding and discrete solution includes the runtime for computing continuous solutions. Area reduction and speedup are obtained by comparing to binary batch.

Large Tuning Step											
Circuit	Binary Batch			Nearest Rounding			Our Discrete Solution				
Name	Area	Yield	CPU	Area	Yield	CPU	Area	Yield	CPU	Speedup	Area Red.
s838	259.9	99.8%	552.7	86.5	90.5%	5.7	105.3	99.5%	242.0	2.3×	59.5%
s1238	222.0	99.9%	135.7	100.9	91.7%	1.4	115.0	99.7%	83.7	1.6×	48.2%
s1423	353.2	99.1%	685.3	229.3	97.5%	52.0	223.1	99.3%	248.5	2.8×	36.8%
s1488	545.9	99.3%	187.2	213.3	97.2%	1.6	214.6	99.4%	86.7	2.2×	60.7%
s5378	303.5	99.7%	935.8	89.8	89.2%	61.4	117.5	99.7%	465.8	2.0×	38.7%
s15850	1297.1	99.3%	4158.5	655.3	90.2%	346.7	778.0	99.5%	2310.2	1.8×	40.1%
s35932	2970.6	99.7%	19532.7	1729.2	95.8%	1024.3	1711.9	99.6%	9844.2	2.0×	57.6%
s38417	2513.1	99.8%	7822.1	1512.9	93.1%	576.8	1578.2	99.2%	4343.6	1.8×	37.2%
s38584	3189.2	99.8%	14102.5	1702.9	91.0%	676.8	1955.2	99.8%	5948.3	2.4×	38.7%

The discretization results significantly depend on the discrete step for the tuning ranges. If one can achieve small tuning precision, rounding from the continuous solution may be good. In contrast, in the situation that one cannot have small tuning

precision, i.e., tuning step is large, our discretization approach is highly desirable. The results are summarized in Table XXI for large tuning step case and Table XXII for small tuning step case.

We make the following observations in large tuning step.

- Nearest rounding often leads to large rounding error, i.e., timing yield is significantly decreased. For s5378, yield is below 90%.
- Since Binary Batch approach is not guided by our continuous solution, the solution quality is quite low compared to other approaches. It often doubles the area compared to the discretization approach. In addition, it takes much longer time to run. This is again due to that continuous solution is not used. For the other two approaches, we only have two choices (i.e., rounding-up or rounding-down) at each tunable element, which means that the search space has been greatly reduced.
- Our discretization approach achieves good balance between solution quality and runtime. Due to our target compensation idea and the effectiveness of LH sampling based Monte Carlo simulations, the exact yield always satisfies the target which is 99.0%. This is not the case for Nearest Rounding. For s1423 and s35932, our approach returns solutions with better yield and smaller area cost compared to Nearest Rounding. As our discretization approach maintains a set of solutions in computation (in both clock and logic rounding), it runs slower than Nearest Rounding approach. However, due to various acceleration techniques, our approach is still more efficient than [14]. Note that Binary Batch is actually a faster version of [14], and our discretization approach is consistently better than Binary Batch in terms of both area and runtime.

We observe the following in small tuning step case.

Table XXII. Discrete Solutions for ISCAS'89 benchmark circuits with small tuning step. Runtime for computing nearest rounding and discrete solution includes the runtime for computing continuous solutions. Area reduction and speedup are obtained by comparing to binary batch.

Small Tuning Step											
Circuit	Binary Batch			Nearest Rounding			Our Discrete Solution				
Name	Area	Yield	CPU	Area	Yield	CPU	Area	Yield	CPU	Speedup	Area Red.
s838	237.4	99.8%	903.5	106.3	97.8%	5.7	104.8	99.5%	275.0	3.3×	55.9%
s1238	213.5	99.5%	151.2	112.8	99.8%	1.4	112.5	99.9%	95.0	1.6×	43.7%
s1423	247.3	99.2%	1072.1	219.5	99.3%	51.9	217.5	99.4%	286.1	3.7×	12.1%
s1488	527.3	99.7%	242.5	212.0	99.1%	1.6	212.5	99.2%	99.9	2.4×	59.7%
s5378	253.1	99.5%	1583.2	113.2	99.3%	61.4	110.1	99.5%	444.7	3.6×	56.5%
s15850	1023.7	99.4%	6255.0	767.1	99.0%	346.0	743.9	99.5%	2525.0	2.5×	27.3%
s35932	2329.0	99.3%	27380.6	1638.0	99.3%	1022.6	1607.2	99.1%	11599.3	2.4×	31.0%
s38417	2175.1	99.6%	11259.7	1452.8	98.2%	575.6	1488.2	99.8%	4774.3	2.4×	31.5%
s38584	2532.5	99.3%	19803.2	1773.2	95.5%	676.3	1892.0	99.7%	6907.1	2.9×	25.3%

- Nearest Rounding obtains good discrete solution in short time, since with smaller tuning step, the continuous solution can be rounded with smaller errors compared to large tuning step case.
- The solution quality of Binary Batch is worst among all algorithms. In addition, it runs slower than large step case since more discrete levels need to be handled.
- Our discretization approach often saves area compared to Nearest Rounding while achieving high timing yield.

In summary, one sees that our continuous unified adaptivity optimization approach is consistently better than optimization on logic or clock signal alone, and it runs very efficiently since no Monte Carlo simulations are involved. In addition, our discretization approach is highly effective and it significantly outperforms Nearest Rounding and Binary Batch.

G. Conclusion

To the best of the authors' knowledge, this work is the first one considering unified adaptivity optimization on logic and clock signals, which saves much area cost compared to optimization on logic or clock signals alone and it does need any assumption on variation distributions. Our continuous optimization is based on a novel linear programming formulation which is efficiently solved by a robust technique where no Monte Carlo simulation is needed. To compute the discrete solution, the continuous solution is used to guide the discretization process to greatly reduce search space. This process involves binary search accelerated dynamic programming, batch based optimization, and Latin Hypercube sampling based fast simulation. Our experimental results demonstrate that up to 50% area cost reduction can be obtained by the unified tuning and the discretization significantly outperforms the alternatives in terms of solution quality and runtime.

CHAPTER VI

CONCLUSION

In the dissertation, various innovative algorithmic techniques are proposed for design and manufacturing closure in nanoscale circuit design. They can be classified into two categories, namely, deterministic optimizations and variation-aware optimizations.

Two deterministic optimizations, namely, buffer insertion and gate sizing, are addressed in the research. For buffer insertion, a new slew buffering formulation is presented and the general slew buffering problem is proved to be NP-hard. Despite this, an ultra-fast dynamic programming algorithm is proposed. It is then extended to handle the difficult case without input slew assumptions, which involves the maximum matching technique. We also propose algorithms for continuous slew buffering and slew buffering with blockage avoidance which makes the approaches ready for practical use. For gate sizing, a new algorithm is proposed to handle discrete gate library in contrast to unrealistic continuous gate library assumed by most existing algorithms. Our approach is a continuous solution guided dynamic programming approach, which integrates the high solution quality of dynamic programming with the short runtime of rounding continuous solution. Our experimental results demonstrate that the new algorithm saves up to 21% area while satisfying the timing constraint compared to the existing alternative.

Two variation-aware optimizations, namely, lithography-driven optimizations and post-silicon tuning-driven optimizations, are addressed in the research. For lithography-driven optimization, three algorithms are proposed for the problem of cell placement considering manufacturability. They are cell flipping algorithm, single row optimization algorithm and multiple row optimization algorthm. These algorithms are based on dynamic programming and graph theoretic approaches, and can provide differ-

ent tradeoff between critical dimension (CD) variation reduction and wirelength increase. Our experimental results on realistic netlists and cell library demonstrate that over 15% CD variation reduction can be obtained in post-OPC stage by the new approaches while only less than 1% additional wire is introduced. For post-silicon tuning-driven optimization, a new algorithm for unified adaptivity optimization on logical and clock signal tuning is proposed. It is based on a novel linear programming formulation which is solved by an advanced robust linear programming technique. The continuous solution is then discretized by binary search accelerated dynamic programming, batch based optimization, and Latin Hypercube sampling based fast simulation. Our experimental results demonstrate that up to 50% area cost reduction can be obtained by the unified optimization compared to optimization on logic or clock alone. In addition, the proposed discretization approach significantly outperforms the alternatives in terms of solution quality and runtime.

As the design flow of VLSI systems becomes increasingly complex, each design step needs to be optimized for efficient design and manufacturing closure. Although this research focuses on some steps, algorithmic techniques proposed in this research have the potential to be applied to similar problems. It is interesting to incorporate these approaches with problem specific knowledge to effectively and efficiently solve other design automation problems.

REFERENCES

[1] Semiconductor Industry Association, *International Technology Roadmap For Semiconductors*, San Jose, California, 2005.

[2] S. Heo, K. Barr, and K. Asanovic, "Reducing power density through activity migration," in *Proc. ACM International Symposium on Low Power Electronics and Design*, Seoul, Korea, August 2003, pp. 217–222.

[3] P. Saxena and N. Menezes and P. Cocchini and D.A. Kirkpatrick, "Repeater scaling and its impact on CAD," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 4, pp. 451–463, April 2004.

[4] J. Cong, "An interconnect centric design flow for nanometer technologies," *Proc. IEEE*, vol. 89, pp. 505–528, April 2001.

[5] P.J. Osler, "Placement driven synthesis case studies on two sets of two chips: hierarchical and flat," in *Proc. ACM International Symposium on Physical Design*, Phoenix, Arizona, April 2004, pp. 190–197.

[6] O. Coudert, "Gate sizing for constrained delay/power/area optimization," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 5, no. 4, pp. 465–472, 1997.

[7] F. Beeftink, P. Kudva, D. Kung, and L. Stok, "Gate-size selection for standard cell libraries," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 1998, pp. 545–550.

[8] M. Mani, A. Devgan, and M. Orshansky, "An efficient algorithm for statistical minimization of total power under timing yield constraints," in *Proc.*

ACM/IEEE Design Automation Conference, Anaheim, California, June 2005, pp. 309–314.

[9] A. Agarwal, K. Chopra, D. Blaauw, and V. Zolotov, "Circuit optimization using statistical static timing analysis," in *Proc. ACM/IEEE Design Automation Conference*, Anaheim, California, June 2005, pp. 321–324.

[10] J. Singh, V. Nookala, Z.-Q. Luo, and S. S. Sapatnekar, "Robust gate sizing by geometric programming," in *Proc. ACM/IEEE Design Automation Conference*, Anaheim, California, June 2005, pp. 315–320.

[11] S.H. Kulkarni, D.M. Sylvester, and D. Blaauw, "A statistical framework for post-silicon tuning through body bias clustering," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 2006, pp. 39–46.

[12] J.W. Tschanz, J.T. Kao, S.G. Narendra, R. Nair, D.A. Antoniadis, A.P. Chandrakasan, and V. De, "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1396–1401, November 2002.

[13] M. Mani, A. Singh, and M. Orshansky, "Joint design-time and post-silicon minimization of parametric yield loss using adjustable robust optimization," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 2006, pp. 19–26.

[14] J.-L. Tsai, L. Zhang, and C.-P. Chen, "Statistical timing analysis driven post-silicon-tunable clock-tree synthesis," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 2005, pp.

575–581.

[15] V. Khandelwal and A. Srivastava, "Variability-driven formulation for simultaneous gate sizing and post-silicon tunability allocation," in *Proc. ACM International Symposium on Physical Design*, Austin, Texas, March 2007, pp. 11–18.

[16] L.P.P.P. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *Proc. IEEE International Symposium on Circuits and Systems*, New Orleans, Louisiana, May 1990, pp. 865–868.

[17] J. Lillis and C.-K. Cheng and T.-T.Y. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE Journal of Solid State Circuits*, vol. 31, no. 3, pp. 437–447, March 1996.

[18] C.J. Alpert and A. Devgan and S.T. Quay, "Buffer insertion for noise and delay optimization," in *Proc. ACM/IEEE Design Automation Conference*, San Francisco, California, June 1998, pp. 362–367.

[19] W. Shi and Z. Li and C. Alpert, "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost," in *Proc. IEEE Asia and South Pacific Design Automation Conference*, Yokohama, Japan, January 2004, pp. 609–614.

[20] W. Shi and Z. Li, "A fast algorithm for optimal buffer insertion," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 879–891, June 2005.

[21] Z. Li, C.N. Sze, C.J. Alpert, J. Hu, and W. Shi, "Making fast buffer insertion even faster via approximation techniques," in *Proc. IEEE Asia and South Pacific Design Automation Conference*, January 2005, pp. 13–18.

[22] Z. Li and W. Shi, "An $O(bn^2)$ time algorithm for optimal buffer insertion with b buffer types," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 484–489, March 2006.

[23] N. Menezes and C.-P. Chen, "Spec-based repeater insertion and wire sizing for on-chip interconnect," in *Proc. IEEE International Conference on VLSI Design*, 1999, pp. 476–483.

[24] C.J. Alpert and A. Devgan and S.T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," in *Proc. ACM/IEEE Design Automation Conference*, New Orleans, Louisiana, June 1999, pp. 479–484.

[25] C.J. Alpert and A.B. Kahng and B. Liu and I. Mandoiu and A. Zelikovsky, "Minimum-buffered routing of non-critical nets for slew rate and reliability control," in *Proc. IEEE/ACM International Conference on Computer Aided Design*, San Jose, California, November 2001, pp. 408–415.

[26] C.J. Alpert and J. Hu and S.S. Sapatnekar and P.G. Villarrubia, "A practical methodology for early buffer and wire resource allocation," in *Proc. ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, June 2001, pp. 189–194.

[27] C.V. Kashyap and C.J. Alpert and F. Liu and A. Devgan, "Closed form expressions for extending step delay and slew metrics to ramp inputs," in *Proc. ACM International Symposium on Physical Design*, Monterey, California, April 2003, pp. 24–31.

[28] H.B. Bakoglu, *Circuits, Interconnects, and Packaging for VLSI*, Boston, MA, Addison Wesley, 1990.

[29] N.H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, pp. 221–223, Boston, MA, Addison Wesley, 1993.

[30] "EinsTimer Users Guide and Language Reference," *Hopewell Junction, NY, IBM Microelectronics Division*, 1995.

[31] T. Feder and R. Motwani, "Clique partitions, graph compression, and speeding-up algorithms," in *Proc. ACM Symposium on Theory of Computing*, New Orleans, Louisiana, May 1991, pp. 123–133.

[32] A.V. Goldberg, "An efficient implementation of a scaling minimum-cost flow algorithm," *Journal of Algorithms*, vol. 22, no. 1, pp. 1–29, January 1997.

[33] J. Hu and C.J. Alpert and S.T. Quay and G. Gandham, "Buffer insertion with adaptive blockage avoidance," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 4, pp. 492–498, April 2003.

[34] J.P. Fishburn and A.E. Dunlop, "TILOS: A posynomial programming approach to transistor sizing," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 1985, pp. 326–328.

[35] C.-P. Chen, C.C.N. Chu, and D.F. Wong, "Fast and exact simultaneous gate and wire sizing by lagrangian relaxation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 7, pp. 1014–1025, July 1999.

[36] S. Boyd, S.J. Kim, D. Patil, and M. Horowitz, "Digital circuit optimization via geometric programming," *Operations Research*, vol. 53, no. 6, pp. 899–932, 2005.

[37] W. Chuang, S. Sapatnekar, and I. Hajj, "Delay and area optimization for discrete gate sizes under double-sided timing constraints," in *Proc. IEEE Custom Integrated Circuits Conference*, San Diego, California, May 1993, pp. 9.4.1–9.4.4.

[38] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. ACM International Conference on Very Large Data Bases*, Edinburgh, Scotland, September 1999, pp. 518–529.

[39] K. Kasamasetty, M. Ketkar, and S. S. Sapatnekar, "A new class of convex functions for delay modeling and its application to the transistor sizing problem," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 779–788, July 2000.

[40] Jeremy Buhler, "Efficient large-scale sequence comparison by locality-sensitive hashing," *Bioinformatics*, vol. 17, no. 5, pp. 419–428, 2001.

[41] A.K.-K. Wong, *Resolution enhancement techniques in optical lithography*, SPIE Press, 2001.

[42] L.-D. Huang and D.F. Wong, "Optical proximity correction (OPC)-friendly maze routing," in *Proc. ACM/IEEE Design Automation Conference*, San Diego, California, June 2004, pp. 186 – 191.

[43] J. Mitra, P. Yu, and D.Z. Pan, "RADAR: RET-aware detailed routing using fast lithography simulations," in *Proc. ACM/IEEE Design Automation Conference*, Anaheim, Californina, June 2005, pp. 369 – 372.

[44] V. Kheterpal, T. Hersan, V. Rovner, D. Motiani, Y. Takagawa, L. Pileggi, and A. Strojwas, "Design methodology for IC manufacturability based on regular logic-bricks," in *Proc. ACM/IEEE Design Automation Conference*, Anaheim, California, June 2005, pp. 353–358.

[45] L. Pileggi, H. Schmit, A.J. Strojwas, P. Gopalakrishnan, V. Kheterpal, A. Koorapaty, C. Patel, V. Rovner, and K.Y. Tong, "Exploring regular fabrics to optimize

the performance-cost trade-off," in *Proc. ACM/IEEE Design Automation Conference*, Anaheim, California, June 2003, pp. 782–787.

[46] L. Liebmann, "Layout impact of resolution enhancement techniques: impediment or opportunity," in *Proc. International Symposium on Physical Design*, Monterey, California, April 2003, pp. 110–117.

[47] P. Gupta, A.B. Kahng, and C.-H. Park, "Detailed placement for improved depth of focus and CD control," in *Proc. IEEE Asia and South Pacific Design Automation Conference*, Shanghai, China, January 2005, pp. 343–348.

[48] S. Sinha, C. Chiang, X. Hong, and Y. Cai, "Efficient process-hotspot detection using range pattern matching," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 2006, pp. 625–632.

[49] D.M. Pawlowski and L. Deng and M.D.-F. Wong, "Boundary-based cellwise OPC for standard-cell layouts," in *Proc. SPIE, Volume 6521, Design for Manufacturability through Design-Process Integration*, 2007, p. 65211O.

[50] C.-W. Sham, E.F.Y. Young, and C. Chu, "Optimal cell flipping in placement and floorplanning," in *Proc. ACM/IEEE Design Automation Conference*, San Francisco, California, July 2006, pp. 1109 – 1114.

[51] S. Hu and J. Hu, "Pattern sensitive placement for manufacturability," in *Proc. ACM International Symposium on Physical Design*, Austin, Texas, March 2007, pp. 27–34.

[52] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, Cambridge, MA, MIT Press, 2nd edition, 2001.

[53] N. Viswanathan and C.C.N. Chu, "Fastplace: Efficient analytical placement

using cell shifting, iterative local refinement and a hybrid net model," in *Proc. ACM International Symposium on Physical Design*, Phoenix, Arizona, April 2004, pp. 26–33.

[54] M. Mani and M. Orshansky, "A new statistical optimization algorithm for gate sizing," in *Proc. International Conference on Computer Design*, San Jose, California, October 2004, pp. 272–277.

[55] A. Singh, M. Mani, and M. Orshansky, "Statistical technology mapping for parametric yield," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 2005, pp. 511–518.

[56] A. Agarwal, K. Chopra, V. Zolotov, and D. Blaauw, "Statistical timing based optimization using gate sizing," in *Proc. ACM/IEEE Design Automation and Test in Europe Conference*, Munich, Germany, March 2005, pp. 400–405.

[57] D. Bertsimas and M. Sim, "The price of robustness," *Operations Research*, vol. 52, no. 1, pp. 35–53, 2004.

[58] V. Nawale and T. Chen, "Optimal useful clock skew scheduling in the presence of variations using robust ILP-formulations," in *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November 2006, pp. 27–32.

[59] A.L. Soyster, "Convex programming with set-inclusive constraints and applications to inexact linear programming," *Operations Research*, vol. 21, pp. 1154–1157, 1973.

[60] K.T. Fang, K. Fang, and L. Runze, *Design and Modelling for Computer Experiments*, Boca Raton, FL, CRC Press, 2005.

[61] S.K. Tiwary, P.K. Tiwary, and R.A. Rutenbar, "Generation of yield-aware pareto surfaces for hierarchical circuit design space exploration," in *Proc. ACM/IEEE Design Automation Conference*, San Francisco, Californina, July 2006, pp. 31–36.

VITA

Shiyan Hu received the B.S. degree in Computer Science and Technology from Beijing University of Aeronautics and Astronautics, China, the M.S. degree in Computer Science from Polytechnic University, Brooklyn, NY, and the Ph.D. degree in Computer Engineering from Texas A&M University.

In 2007, he worked as a graduate research intern at the IBM Austin Research Lab. His research interests are primarily on VLSI Computer-Aided Design including buffer insertion, routing, gate sizing, variation-aware optimization, and design for manufacturability. His mailing address is Department of Electrical and Computer Engineering, Mail Stop 3128, Texas A&M University, College Station, TX, 77843-3128.

The typist for this thesis was Shiyan Hu.